O'REILLY®

TURING 图灵程序设计丛书



Node与Express开发

WEB DEVELOPMENT WITH NODE AND EXPRESS

[美] Ethan Brown 著
吴海星 苏文 译

人民邮电出版社
POSTS & TELECOM PRESS

译者介绍

吴海星

“这个译者很懒，他什么都不想说。”新浪微博：@数据水墨。图灵社区ID：海兴。

苏文

八零后，现居北京，从事互联网金融行业，尘世中一个辛勤的小码农。

TURING 图灵程序设计丛书

Node与Express开发

Web Development with Node and Express
Leveraging the JavaScript Stack

[美] Ethan Brown 著

吴海星 苏文 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北 京

图书在版编目（ＣＩＰ）数据

Node与Express开发 / (美) 布朗 (Brown,E.) 著 ；
吴海星，苏文译. -- 北京 ：人民邮电出版社，2015.2
（图灵程序设计丛书）
ISBN 978-7-115-38033-3

Ⅰ. ①N… Ⅱ. ①布… ②吴… ③苏… Ⅲ. ①JAVA语
言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第306213号

内 容 提 要

本书涵盖 Express 4.0，系统讲解了使用 Express 开发动态 Web 应用的流程和步骤。作者不仅向读者讲授了开发公共站点及 REST API 的基础知识，同时还讲解了构建单页、多页及混合 Web 应用的规划方式及最佳实践。具体而言，本书内容包括创建模板、请求及响应对象、中间件、URL 路由、模拟测试、文档数据库、社交媒体集成、启动与维护应用、调试，等等。

本书适合所有前端和后端开发人员阅读。

◆ 著　　　　[美] Ethan Brown
译　　　　吴海星　苏　文
责任编辑　岳新欣
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800 × 1000　1/16
印张：17.25
字数：355千字　　　　2015年 2 月第 1 版
印数：1 - 3 500册　　　2015年 2 月北京第 1次印刷
著作权合同登记号　图字：01-2014-7515号

定价：69.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

谨以此书献给我的家人。

父亲 Tom 让我爱上了工程，母亲 Ann 让我爱上了写作，姐姐 Meris 则一直陪伴着我。

目录

序 …… XIV

前言 …… XV

第 1 章　初识 Express …… 1

1.1　JavaScript 革命 …… 1
1.2　初识 Express …… 2
1.3　Express 简史 …… 3
1.4　升级到 Express 4.0 …… 4
1.5　Node：一种新型 Web 服务器 …… 4
1.6　Node 的生态系统 …… 5
1.7　授权 …… 6

第 2 章　从 Node 开始 …… 8

2.1　获取 Node …… 8
2.2　使用终端 …… 9
2.3　编辑器 …… 10
2.4　npm …… 11
2.5　用 Node 实现的简单 Web 服务器 …… 12
2.5.1　Hello World …… 12
2.5.2　事件驱动编程 …… 13
2.5.3　路由 …… 13
2.5.4　静态资源服务 …… 14
2.6　走向 Express …… 16

第 3 章　省时省力的 Express……17
3.1　脚手架……17
3.2　草地鹨旅行社网站……18
3.3　初始步骤……18
3.3.1　视图和布局……21
3.3.2　视图和静态文件……24
3.3.3　视图中的动态内容……24
3.4　小结……25
第 4 章　工欲善其事，必先利其器……26
4.1　最佳实践……26
4.2　版本控制……27
4.3　针对本书如何使用 Git……27
4.3.1　如果你要自己动手……28
4.3.2　如果你要使用官方存储库……29
4.4　npm 包……29
4.5　项目元数据……31
4.6　Node 模块……31
第 5 章　质量保证……33
5.1　QA：值得吗……34
5.2　逻辑与展示……35
5.3　测试的类型……35
5.4　QA 技术概览……35
5.5　运行你的服务器……36
5.6　页面测试……36
5.7　跨页测试……40
5.8　逻辑测试……43
5.9　去毛……43
5.10　链接检查……44
5.11　用 Grunt 实现自动化……44
5.12　持续集成……46
第 6 章　请求和响应对象……48
6.1　URL 的组成部分……48
6.2　HTTP 请求方法……49
6.3　请求报头……50

6.4 响应报头······50
6.5 互联网媒体类型······51
6.6 请求体······51
6.7 参数······51
6.8 请求对象······51
6.9 响应对象······53
6.10 获取更多信息······55
6.11 小结······56
6.11.1 内容渲染······56
6.11.2 处理表单······57
6.11.3 提供一个 API······58
第 7 章 Handlebars 模板引擎······60
7.1 唯一一条绝对规则······61
7.2 选择模板引擎······61
7.3 Jade：不走寻常路······62
7.4 Handlebars 基础······63
7.4.1 注释······64
7.4.2 块级表达式······64
7.4.3 服务器端模板······66
7.4.4 视图和布局······67
7.4.5 在 Express 中使用（或不使用）布局······69
7.4.6 局部文件······69
7.4.7 段落······71
7.4.8 完善你的模板······72
7.4.9 客户端 Handlebars······73
7.5 小结······75
第 8 章 表单处理······76
8.1 向服务器发送客户端数据······76
8.2 HTML 表单······76
8.3 编码······77
8.4 处理表单的不同方式······78
8.5 Express 表单处理······79
8.6 处理 AJAX 表单······81
8.7 文件上传······83
8.8 jQuery 文件上传······85

第 9 章　Cookie 与会话 88
9.1　凭证的外化 89
9.2　Express 中的 Cookie 90
9.3　检查 Cookie 91
9.4　会话 92
9.4.1　内存存储 92
9.4.2　使用会话 93
9.5　用会话实现即显消息 93
9.6　会话的用途 95
第 10 章　中间件 96
10.1　常用中间件 100
10.2　第三方中间件 102
第 11 章　发送邮件 103
11.1　SMTP、MSA 和 MTA 103
11.2　接收邮件 104
11.3　邮件头 104
11.4　邮件格式 104
11.5　HTML 邮件 105
11.6　Nodemailer 105
11.6.1　发送邮件 106
11.6.2　将邮件发送给多个接收者 107
11.7　发送批量邮件的更佳选择 108
11.8　发送 HTML 邮件 108
11.8.1　HTML 邮件中的图片 108
11.8.2　用视图发送 HTML 邮件 109
11.8.3　封装邮件功能 111
11.9　将邮件作为网站监测工具 112
第 12 章　与生产相关的问题 113
12.1　执行环境 113
12.2　环境特定配置 114
12.3　扩展你的网站 115
12.3.1　用应用集群扩展 116
12.3.2　处理未捕获的异常 118
12.3.3　用多台服务器扩展 121

12.4 网站监控 122
12.4.1 第三方正常运行监控 122
12.4.2 应用程序故障 122
12.5 压力测试 123

第 13 章 持久化 124
13.1 文件系统持久化 124
13.2 云持久化 126
13.3 数据库持久化 126
13.3.1 关于性能 127
13.3.2 设置 MongoDB 127
13.3.3 Mongoose 128
13.3.4 使用 Mongoose 连接数据库 128
13.3.5 创建模式和模型 129
13.3.6 添加初始数据 130
13.3.7 获取数据 131
13.3.8 添加数据 133
13.3.9 用 MongoDB 存储会话数据 134

第 14 章 路由 137
14.1 路由和 SEO 139
14.2 子域名 139
14.3 路由处理器是中间件 140
14.4 路由路径和正则表达式 141
14.5 路由参数 142
14.6 组织路由 143
14.7 在模块中声明路由 143
14.8 按逻辑对处理器分组 144
14.9 自动化渲染视图 145
14.10 其他的路由组织方式 146

第 15 章 REST API 和 JSON 147
15.1 JSON 和 XML 148
15.2 我们的 API 148
15.3 API 错误报告 149
15.4 跨域资源共享 150
15.5 我们的数据存储 150
15.6 我们的测试 151

15.7 用 Express 提供 API……152
15.8 使用 REST 插件……153
15.9 使用子域名……155

第 16 章 静态内容……157

16.1 性能方面的考虑……158
16.2 面向未来的网站……158
16.2.1 静态映射……159
16.2.2 视图中的静态资源……160
16.2.3 CSS 中的静态资源……161
16.3 服务器端 JavaScript 中的静态资源……162
16.4 客户端 JavaScript 中的静态资源……163
16.5 提供静态资源……164
16.6 修改静态内容……165
16.7 打包和缩小……165
16.8 关于第三方库……170
16.9 QA……170
16.10 小结……171

第 17 章 在 Express 中实现 MVC……173

17.1 模型……174
17.2 视图模型……175
17.3 控制器……177
17.4 小结……179

第 18 章 安全……180

18.1 HTTPS……180
18.1.1 生成自己的证书……181
18.1.2 使用免费的证书颁发机构……182
18.1.3 购买证书……182
18.1.4 对你的 Express 应用启用 HTTPS……184
18.1.5 关于端口的说明……185
18.1.6 HTTPS 和代理……185
18.2 跨站请求伪造……187
18.3 认证……187
18.3.1 认证与授权……188
18.3.2 密码的问题……188
18.3.3 第三方认证……188

18.3.4 把用户存在数据库中 189
18.3.5 认证与注册和用户体验 190
18.3.6 Passport 190
18.3.7 基于角色的授权 199
18.3.8 添加更多认证提供者 200
18.4 小结 201

第 19 章 集成第三方 API 202

19.1 社交媒体 202
19.1.1 社交媒体插件和站点性能 202
19.1.2 搜索推文 203
19.1.3 渲染推文 206
19.2 地理编码 209
19.2.1 用谷歌的地理编码 209
19.2.2 对你的数据做地理编码 210
19.2.3 显示地图 213
19.2.4 提升客户端性能 215
19.3 天气数据 216
19.4 小结 217

第 20 章 调试 218

20.1 调试的首要原则 218
20.2 利用好 REPL 和控制台 219
20.3 利用 Node 内置的调试器 220
20.4 Node 探查器 220
20.5 调试异步函数 223
20.6 调试 Express 224

第 21 章 正式启用 226

21.1 域名注册和托管服务 226
21.1.1 域名系统 227
21.1.2 安全 227
21.1.3 顶级域名 228
21.1.4 子域名 229
21.1.5 域名服务器 229
21.1.6 托管 230
21.1.7 部署 233
21.2 小结 236

第 22 章　维护 ······ 237
22.1 维护的原则 ······ 237
22.1.1 有长远规划 ······ 237
22.1.2 使用源码控制系统 ······ 239
22.1.3 使用问题追踪系统 ······ 239
22.1.4 良好的卫生习惯 ······ 239
22.1.5 不要拖延 ······ 239
22.1.6 做常规的 QA 检查 ······ 240
22.1.7 监测分析 ······ 240
22.1.8 性能优化 ······ 240
22.1.9 潜在用户追踪优先 ······ 241
22.1.10 防止出现“不可见的”错误 ······ 242
22.2 代码重用及重构 ······ 243
22.2.1 私有 npm 库 ······ 243
22.2.2 中间件 ······ 244
22.3 小结 ······ 246
第 23 章　其他资源 ······ 247
23.1 在线文档 ······ 247
23.2 期刊 ······ 248
23.3 Stack Overflow ······ 248
23.4 为 Express 做贡献 ······ 250
23.5 小结 ······ 252
关于封面 ······ 253
关于作者 ······ 254

序

JavaScript、Node 和 Express 的组合是 Web 团队的理想选择，这个强大的、可快速部署的技术栈得到了开发社区和大公司的广泛认可。

构建优秀的 Web 应用程序和寻找优秀的 Web 开发人员都不容易。优秀的应用程序需要出色的功能、用户体验，并能提升业务能力：快速交付、部署和提供支持，且成本合适。Express 提供了较低的总体拥有成本和较快的上市时间，这在商业世界中至关重要。如果你是一名 Web 开发人员，至少也会用到一些 JavaScript，但你也可以大量使用它。Ethan Brown 在本书中向你展示了如何大量使用它，而且多亏 Node 和 Express，做到这一点并不难。

Node 和 Express 就像发射 JavaScript 希望之银弹的机关枪。

JavaScript 是应用最广泛的客户端脚本语言。与 Flash 不同，所有主流 Web 浏览器都支持 JavaScript。你在 Web 上看到的很多动人的动画和切换效果都是以这一技术为基础的。实际上，如果你想充分发挥现代浏览器的功能，不用 JavaScript 几乎是不可能的。

JavaScript 的一个问题是总容易受到草率编程的拖累。Node 生态系统提供的框架、库和工具改变了这种状况，它们可以加速开发，鼓励良好的编程习惯。这能帮我们更快地把好应用推向市场。

我们现在有了一个由大公司支持的伟大编程语言，它易于使用，专为现代浏览器而设计，并且在客户端和服务器端都有优秀的框架和库。我管它叫一场革命。

——Steve Rosenbaum

Pop Art 公司总裁兼首席执行官

前言

读者对象

很明显，本书是给想要用 JavaScript、Node 和 Express 创建 Web 应用程序（传统网站、REST API，或者介于两者之间的任何东西）的程序员准备的。Node 开发令人兴奋的一面是它已经吸引了全新的程序员受众。JavaScript 的可用性和灵活性吸引了来自世界各地的自学成才的程序员。在计算机科学的历史中，编程还从没有如此容易过。学习编程的在线资源的品质和数量（以及遇到困难时获取的帮助）真的令人惊讶和鼓舞人心。所以对于那些新（可能是自学）的程序员，我表示欢迎。

当然，还有像我这样已经做过一段时间编程的程序员。与同时代的很多程序员一样，我也是从汇编和 BASIC 开始的，然后经历了 Pascal、C++、Perl、Java、PHP、Ruby、C、C# 和 JavaScript。上大学时，我接触过更加小众的语言，比如 ML、LISP 和 PROLOG。这些语言很多都接近我的理想选择，但没有一个像 JavaScrip 这样让我觉得前景如此光明。所以这本书也是给像我这样的程序员写的，他们经验丰富，可能对特定技术的认识更富哲理。

你不一定要有 Node 方面的经验，但应该有一些 JavaScript 经验。如果你刚接触编程，建议你到 Codecademy（http://www.codecademy.com/tracks/javascript）上看看。如果你是有经验的程序员，推荐你看看 Douglas Crockford 的 *JavaScript: The Good Parts*（O'Reilly，http://book.douban.com/subject/2994925/）。本书中的例子可以在 Node 支持的任何系统（包括 Windows、OS X 和 Linux）上使用。这些示例主要面向命令行（终端）用户，所以你应该熟悉你所使用的系统的终端。

最重要的是，本书是为那些跃跃欲试的程序员准备的。他们对互联网的未来感到兴奋，并且想参与其中。他们对学新东西、新技术和 Web 开发的新方式感到兴奋。亲爱的读者，如果你没有兴奋感，我希望你读完本书时能有这种感觉……

内容安排

第 1 章和第 2 章将会介绍 Node 和 Express，以及你在整本书中都会用到的一些工具。在第 3 章和第 4 章中，你将开始用 Express 搭建一个示例网站的骨架，这个网站也是贯穿本书始终的例子。

第 5 章讨论测试和 QA。第 6 章介绍 Node 中一些更重要的结构，以及 Express 如何扩展和使用它们。第 7 章讲解模板（用 Handlebars），为使用 Express 搭建有用的网站打下基础。第 8 章和第 9 章介绍 cookies、会话和表单处理器，这些是用 Express 搭建基本可用的网站需要了解的基础知识。

第 10 章深入探讨中间件，这是 Connect（Express 的主要组件之一）的核心概念。第 11 章解释如何用中间件从服务器发送电子邮件，并讨论邮件的安全和布局问题。

第 12 章提供产品问题的预览。即便到这一阶段，你也没有掌握搭建产品环境中的网站所需的全部信息，但现在就考虑产品环境可以让你在将来免受巨大的痛苦。

第 13 章讨论持久化，内容主要围绕 MongoDB（一种领先的文档数据库）展开。

第 14 章介绍 Express 中路由的细节（URL 如何映射到内容）。第 15 章深入探讨如何用 Express 编写 API。第 16 章介绍提供静态内容的细节，并重点介绍性能最大化。第 17 章重申流行的模型 – 视图 – 控制器（MVC）范式，以及它如何融入 Express。

第 18 章讨论安全：如何在程序中搭建认证和授权（重点介绍如何使用第三方认证），以及如何通过 HTTPS 运行网站。

第 19 章解释如何集成第三方服务。所用的例子是 Twitter、谷歌地图和 Weather Underground。

第 20 章和第 21 章让你准备好迎接重要的日子：网站的正式启用。内容包括调试（以便你能在启用网站前找出所有的缺陷）以及启用网站的流程。第 22 章谈及下一个重要（但经常被忽略）的阶段：维护。

第 23 章是本书的结尾，指出若想继续深入学习 Node 和 Express 可参考哪些其他资源，以及到哪里去寻求帮助。

示例网站

从第 3 章开始，会有一个贯穿全书的例子：草地鹨旅行社网站。我刚从里斯本旅行回来，对旅行还念念不忘，所以我选的示例网站是虚构的我家乡俄勒冈州一家旅行社（西部草地鹨是俄勒冈州的州鸟）。草地鹨旅行社允许旅行者跟本地的“业余导游”联系，它还跟其

他公司合作提供自行车和摩托车租赁及本地游服务。此外，它还维护一个当地景点的数据库，配有历史和位置感知服务。

跟所有教学示例一样，草地鹨旅行社网站是瞎编的，但这个例子涉及很多在现实世界中也会遇到的挑战：第三方组件集成、地理位置服务、电子商务、性能和安全。

因为本书的重点是后端基础设施，所以示例网站不是完整的，它仅仅作为一个假想示例提供例子的深度和上下文。如果你在搭建自己的网站，可以用草地鹨旅行社作为模板。

排版约定

本书使用了下述排版约定。

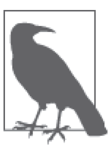
- 楷体
  标示新术语。

- 等宽字体
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体
  表示应该由用户直接输入的命令或其他文本。



该图标表示提示或建议。



该图标表示普通的注记。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/EthanRBrown/web-development-with-node-and-express 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Web Development with Node and Express* by Ethan Brown (O'Reilly). Copyright 2014 Ethan Brown, 978-1-491-94930-6."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的意见和疑问发送给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://bit.ly/web_dev_node_express

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

我生命中的很多人都为本书的出版做出了贡献。如果没有那些触及我的生命并影响我的人，我就不可能完成本书。

我想首先感谢 Pop Art 的每一位。我在 Pop Art 的时光不仅重燃了我对工程的热情，还让我从每个人身上学到了很多。如果没有他们的支持，本书将无法完成。感谢 Steve Rosenbaum 创造了这样一个振奋人心的工作场所。感谢 Del Olds 将我带到 Pop Art，让我感受到大家的热情，并荣幸地成为一个领导者。感谢 Paul Inman 的无私支持以及对工程的热忱。感谢 Tony Alferez 的热情支持，并帮我抽出时间写作，让我没有影响 Pop Art 的工作。最后感谢我共事过的所有优秀工程师，让我专注于我的事业，他们是 John Skelton、Dylan Hallstrom、Greg Yung、Quinn Michael 和 CJ Stritzel。

Zach Mason，感谢你的激励。这本书可能不是你的 *The Lost Books of the Odyssey*，但它是我的。我不知道如果没有你这个例子，我还能不能如此大胆。

我的一切都归功于我的家庭。我无法想象还有比我父母给我的更好的、充满爱的教育，我妹妹身上也体现了他们良好的教育。

非常感谢 Simon St. Laurent 给我这次机会。感谢 Brian Anderson 进行了平稳又出色的编辑。感谢 O'Reilly 的每一位，他们既专注又热情。感谢 Jennifer Pierce、Mike Wilson、Ray Villalobos 和 Eric Elliot，他们进行了彻底且富有建设性的技术审查。

Katy Roberts 和 Hanna Nelson 对我“冒昧”的提案提供了宝贵的反馈和建议，使本书成为可能。非常感谢你们两位！感谢 Chris Cowell-Shah 对“质量保证”一章的精彩反馈。

最后，感谢我亲爱的朋友们，没有你们我肯定已经疯了。Byron Clayton、Mark Booth、Katy Roberts 和 Sarah Lewis，我不可能有比你们再好的朋友了。还要感谢 Vickey 和 Judy。我爱你们。

第 1 章

初识Express

1.1 JavaScript革命

在介绍本书的主要内容之前，我首先介绍一些背景知识和历史沿革，也就是谈一谈 JavaScript 和 Node。

JavaScript 的时代真的来临了。最开始它只是一种粗陋的客户端脚本语言，但现在它不仅是客户端普遍使用的脚本语言，甚至还因为 Node 的出现最终成为了服务器端脚本语言。

全部由 JavaScript 组成的技术栈前景非常明朗：不再需要环境切换！你再也不需要从 JavaScript 的思维模式切换到 PHP、C#、Ruby 或 Python（或其他任何服务器端语言）。此外，它还让前端工程师一跃进入了服务器端编程领域。当然，这并不是说服务器端编程只和语言有关，仍然有很多东西需要学习。但有了 JavaScript，至少语言不再是障碍了。

这本书是为所有看到 JavaScript 技术栈前景的人而写的。你或许是一个想积累后端开发经验的前端工程师，或许是一个经验丰富的后端开发人员，像我一样想把 JavaScript 作为自己的服务器端编程语言选择之一。

如果你和我一样做了很长时间的软件工程师，一定见证过很多语言、框架和 API 的兴起。其中有些已经销声匿迹，还有些已经陈旧过时了。你或许会对自己快速学习新语言、新系统的能力引以为傲。每遇到一种新语言，你都会觉得更熟悉一些：有些是在大学学习的语言里见过，有些是在你几年前的工作中见过。持有这种观点当然会感觉很好，但也会让人感到厌倦。有时你只是想完成某件事情，而不想为此再学习一种全新的技术，或者重新使用尘封了几个月甚至几年的技术。

起初，JavaScript 看起来并不可能胜出，当时我的想法亦是如此。如果三年前有人说我不仅会选择 JavaScript 作为我的语言，还会就此写一本书，我一定会认为他是个疯子。对 JavaScript，我曾经抱有和大家一样的偏见，觉得它只是个“玩具”，是给业余选手和一知半解的人随意使用的语言。老实说，JavaScript 确实降低了业余选手进入的门槛，也有很多充斥着各种问题的 JavaScript 代码，这损坏了 JavaScript 的名声。用句通俗的话说，即“不是游戏太差，而是玩家太烂”。

很可惜，人们对 JavaScript 持有这种偏见，这使得人们没能发现这门语言的强大、灵活和优雅。许多人现在才刚刚开始认真看待 JavaScript，而这门语言在 1996 年前后就已经出现了（尽管很多有吸引力的特性是在 2005 年加上的）。

因为你已经在阅读这本书，所以你应该没有那种偏见。或许是像我一样，有偏见的阶段已经过去了，也或许是从一开始根本就没有对它抱有偏见。无论是哪种情况，你都是幸运的，我殷切地期待着向你介绍 Express，而 Express 这种技术正是由于一种令人愉悦又惊喜的语言才成为可能。

2009 年，人们早已经认识到 JavaScript 作为浏览器脚本语言非常强大，具有很强的表现能力，这时，Ryan Dahl 看到了 JavaScript 作为服务器端语言的潜力，于是 Node 诞生了。这是一个互联网技术生机勃勃的时代。Ruby（和 RoR）吸收了学院派计算机科学的一些伟大思想，并结合了自有的一些新想法，推出了一种更快捷的网站及 Web 应用程序构建方式。微软也通过奋勇作战在互联网时代争得了一席之地，借助 .NET 取得了惊人的成就，它不仅借鉴了 Ruby 和 JavaScript 的优点，还从 Java 犯的错误中吸取了经验，并充分吸收了学术殿堂中的精髓。

徜徉在互联网技术中令人感到兴奋，到处都是令人惊奇的新想法（或者复兴的旧思想）。现在的创新精神和新鲜事物比过去的这许多年要更强、更多。

1.2　初识Express

Express 网站上是这样介绍 Express 的：“精简的、灵活的 Node.js Web 程序框架，为构建单页、多页及混合的 Web 程序提供了一系列健壮的功能特性。”这究竟是什么意思呢？下面我们来逐一解读一下。

- 精简

  这是 Express 最吸引人的特性之一。框架开发者经常会忘掉“少即是多”这一基本原则。Express 的哲学是在你的想法和服务器之间充当薄薄的一层。这并不意味着它不够健壮，或者没有足够的有用特性，而是尽量少干预你，让你充分表达自己的思想，同时提供一些有用的东西。

- *灵活*

  Express 哲学中的另一个关键点是可扩展。Express 提供了一个非常精简的框架，你可以根据自己的需要添加 Express 功能中的不同部分，替换掉不能满足需要的部分。这种做法很新鲜。很多框架把什么都给你了，一行代码还没写，你拥有的就已经是一个臃肿、神秘而复杂的项目了。通常，你的第一项任务就是把不需要的功能砍掉，或者替换掉不能满足需求的功能。Express 则采取了截然不同的方式，让你在需要时才去添加东西。

- *Web程序框架*

  这里需要琢磨一下语义了。什么是 Web 程序？这意味着 Express 就不能做出网站或者网页了吗？不，网站是 Web 程序，网页也是 Web 程序。但 Web 程序的含义不止这些，它还可以向其他 Web 程序提供功能（还有别的）。一般而言，“程序”是具有功能的，它不止是内容的静态集合（尽管这也是非常简单的 Web 程序）。尽管现在“程序”（在你的设备本地运行的东西）和“网页”（通过网络为你的设备服务的东西）之间有明显的界限，但这种界限渐渐变得模糊了，这要感谢 PhoneGap 这样的项目，同时也要感谢微软允许 HTML5 像本地应用程序一样在桌面上运行。不难想象，几年之内程序和网站之间的界限将不复存在。

- *单页Web程序*

  单页 Web 程序是比较新颖的想法。不像之前的网站，用户每次访问不同的页面都要发起网络请求，单页 Web 程序把整个网站（或很大一部分）都下载到客户端浏览器上。经过初始下载后，用户访问不同页面的速度更快了，因为几乎不需要或者只要很少的服务端通信。单页程序的开发可以使用 Angular 或 Ember 等流行框架，Express 跟它们都配合得很好。

- *多页和混合的Web程序*

  多页 Web 程序是更传统的方式。网站上的每个页面都是通过向服务器发起单独的请求得到的。这种方式确实比较传统，但这并不意味着它没有优点，或者说单页程序更好。只是现在有更多选择了，你可以决定哪些内容应该作为单页程序提供，哪些应该通过不同的请求提供。“混合”说的就是同时使用这两种方式的网站。

如果你还是很困惑 Express 究竟是什么，不用担心。有时候只管把某些东西拿来用就好了，不用先理解它是什么，本书将教你如何用 Express 开发 Web 程序。

1.3 Express简史

Express 的缔造者 TJ Holowaychuk 说 Express 是在 Sinatra 的启发下创建的，后者是一个基于 Ruby 的框架。Express 借鉴一个在 Ruby 上构建的框架并不奇怪：Ruby 致力于让 Web 开发变得更快、更高效、更可维护，并衍生了大量的 Web 开发方式。

除了 Sinatra，Express 跟 Connect 也有非常紧密的联系，Connect 是一个 Node 的“插件”库。Connect 创造了“中间件”（middleware）这个术语来描述插入式的 Node 模块，它能在不同程度上处理 Web 请求。在版本 4.0 之前，Express 一直是绑定 Connect 的；在版本 4.0 中，Connect（以及除 `static` 之外的所有中间件）被去掉了，以便这些中间件可以各自独立升级。

> Express 从 2.x 升级到 3.0 时做了大量的改写，从 3.x 到 4.0 时也是这样。本书会重点介绍版本 4.0。

1.4　升级到Express 4.0

如果你用过 Express 3.0，知道可以毫不费力地升级到 Express 4.0 应该会很高兴。如果你刚接触 Express，可以直接跳过这一节。对于用过 Express 3.0 的读者，请注意以下几个重点。

- Connect 已经从 Express 中去掉了，所以除了 `static` 中间件，你需要自己安装相应的开发包（即 `connect`）。与此同时，Connect 将一些中间件移到了它自己的包内，所以你可能要在 npm 上搜一下，看看你需要的中间件到哪去了。
- `body-parser` 现在有自己的包了，它不再包含 `multipart` 中间件，因而也关闭了一个重大的安全漏洞。现在可以放心使用 `body-parser` 中间件了。
- 不必再将 Express router 链接到程序里。所以应该从已有的 Express 3.0 中去掉 `app.use(app.router)`。
- `app.configure` 被去掉了，只要检查 `app.get(env)`（用 `switch` 或 `if` 语句）就可以取代该方法。

更多细节请参阅官方迁移指南（https://github.com/strongloop/express/wiki/Migrating-from-3.x-to-4.x）。

Express 是一个开源项目，主要还是由 TJ Holowaychuk 开发及维护。

1.5　Node：一种新型Web服务器

从某种角度看，Node 跟其他流行的 Web 服务器，比如微软的互联网信息服务（IIS）或 Apache，有很多共同点。然而更有趣的是探究它的不同之处，所以我们先从讨论它的不同开始。

Node 实现 Web 服务器的方式跟 Express 很像，也非常精简。Node 的搭建和配置非常容易，不像 IIS 或 Apache 要花费多年的时间才能掌握。但要让 Node 服务器在生产环境中发挥出

最优性能，进行调优也绝非易事，只不过是配置选项更简单，也更直接了。

Node 和传统的 Web 服务器之间的另一个主要区别是：Node 是单线程的。乍一看可能觉得这是一种倒退。但事实证明，这是天才之举。单线程极大地简化了 Web 程序的编写，如果你需要多线程程序的性能，只需启用更多的 Node 实例，就可以得到多线程的性能优势。精明的读者可能会觉得我这是在放烟幕弹。毕竟，通过服务器并行（相对于程序的并行）的多线程只是把复杂性转移了，并没有消除它啊？也许吧，但依我之见，它是把复杂性放到了它应该存在的地方。更进一步说，随着云计算的日益流行，以及将服务器当作普通商品看待的趋势越来越明显，这种方式也变得更有意义了。IIS 和 Apahce 确实强大，并且它们的设计目标也是要榨取如今强大的硬件设施的最后一点性能。但那是需要付出代价的，即它需要相当专业的设置和调优才能榨取那种性能。

至于编写程序的方式，相较于 .NET 或 Java 程序，Node 程序更像 PHP 或 Ruby。尽管 Node 所用的 JavaScript 引擎（谷歌的 V8）确实会将 JavaScript 编译为本地机器码（更像 C 或 C++），但这一操作是透明的 [1]，所以从用户的角度来看，它表现的还是像纯粹的解释型语言一样。没有单独的编译步骤，这减少了维护和部署的麻烦。你所要做的只是更新 JavaScript 文件，然后你的修改就自动生效了。

Node 程序的另一个好处是它的平台无关性。它不是第一个或唯一的平台无关的服务器技术，但平台无关的水平真的是良莠不齐。例如，你可以借助 Mono 在 Linux 上运行 .NET 程序，但这个过程会很痛苦。同样，你可以在 Windows 服务器上运行 PHP 程序，但一般不像在 Linux 机器上设置那么容易。另一方面，在所有主流操作系统（Windows、OS X 和 Linux）上设置 Node 都易如反掌，并且协作也很容易。在网站设计团队中，经常会同时出现 PC 和 Mac。某些平台，比如 .NET，对经常使用 Mac 的前端开发人员和设计师来说是个挑战，会极大地影响协作性和工作效率。用几分钟（甚至几秒钟）的时间在任意一个操作系统上构建一个可运行服务器的梦想终于实现了。

1.6 Node的生态系统

当然，Node 处于这个技术栈的核心位置。就是它让 JavaScript 从浏览器中分离出来，得以在服务器上运行，进而可以使用 JavaScript 写成的框架（比如 Express）。另外一个重要的组件是数据库，这将在第 13 章中进行详细介绍。除了最简单的 Web 程序，所有的程序都需要数据库，并且 Node 生态系统中的数据库更多。

所有主流关系型数据库（MySQL、MariaDB、PostgreSQL、Oracle、SQL Server）的接口都有，这一点并不奇怪，因为忽视那些已经成熟的“巨无霸”太不明智了。然而 Node 开发的出现带动了一种新式的数据库存储方式，这种方式被称为“NoSQL 数据库”。用否定的

注 1：通常被称作“即时”编译。

方式来下定义有时并不恰当，所以我们更准确地称之为“文档数据库”或“键 / 值对数据库”。它们提供了一种概念上更简单的数据存储方式。这种数据库有很多，但 MongoDB 是其中的佼佼者，也是我们要在本书中使用的数据库。

因为构建一个功能性网站要借助很多种技术，因此衍生了一种用来描述网站构建基础“技术栈”的缩略语。比如说，Linux、Apache、MySQL 和 PHP 被称为 LAMP 栈。MongoDB 的工程师 Valeri Karpov 发明了一个缩略语 MEAN，指代 Mongo、Express、Angular 和 Node。尽管它确实朗朗上口，却有其局限性：可选的数据库和应用程序框架有很多，MEAN 无法体现这个生态系统的多样性（它还漏掉了一个我认为非常重要的组件：模板引擎）。

发明一个兼容并包的缩略语是一个有趣的事情。其中无可替代的组件当然是 Node。尽管还有其他的服务器端 JavaScript 容器，但 Node 是其中的执牛耳者。尽管 Express 在主导地位上接近 Node，但它也不是唯一可用的 Web 程序框架。另外两个通常来说对 Web 程序开发必不可少的组件是数据库服务器和模板引擎（模板引擎提供了 PHP、JSP 或 Razor 自带的功能：将代码和标记输出无缝结合起来）。对于最后两种组件而言，没有明显的领跑者，我认为对此加以限制有害无益。

将所有这些技术结合到一起的是 JavaScript，所以为了做到兼容并包，我将其称为“JavaScript 技术栈”。对于本书而言，即指 Node、Express 和 MongoDB。

1.7　授权

在开发 Node 程序时，你可能会发觉自己要比以往更加关注授权问题（我肯定是这样）。Node 生态系统的美好也体现在大量可用的开发包上。然而那些包都有其自身的授权，甚至更糟，每个包可能还要依赖其他包，也就是说要明白你写的程序各部分的授权是很难的。

然而也存在一些好消息。Node 开发包中最常见的是 MIT 授权，它是毫不费力的许可，几乎允许你做任何想做的事情，包括把开发包放到闭源的软件中。然而，你不能假定使用的所有包都是 MIT 授权。

> npm 中有几个包会试图帮你确定项目中每个依赖项的授权。在 npm 中搜索 license-sniffer 或 license-spelunker。

尽管最常见的授权是 MIT，但你可能也会遇到下面这几种授权。

- GNU通用公共授权（GPL）
  GPL 是非常流行的开源授权，它为保证软件的自由做了精巧的构思。这意味着如果你

在项目中用了 GPL 授权的代码，那么你的项目必须也是 GPL 授权的。这自然也就意味着你的项目不能是闭源的。

- Apache 2.0
  这个授权像 MIT 一样，你可以为自己的项目使用不同的授权，包括闭源的授权。然而，你必须对那些使用 Apache 2.0 授权的组件做出声明。

- 伯克利软件分发（BSD）
  与 Apache 类似，这个授权允许你为自己的项目使用任何授权，只是你声明使用了 BSD 授权的组件。

> 软件有时是双授权的（有两种不同的授权）。一个非常常见的理由是允许软件用在 GPL 项目和有更多许可授权的项目中。（对于用在 GPL 软件中的组件而言，这个组件也必须是 GPL 授权的。）我在自己的项目中也经常使用这一授权方案：GPL 和 MIT 双授权。

最后，如果你在编写自己的包，你应该做个善良的人，选一个授权并在文档中正确声明。对于一个开发人员来说，没有什么比深挖源码才能确定所用开发包的授权更恐怖的了，或者更糟的情况是，发现它根本没有授权。

第2章

从Node开始

如果你从来没接触过 Node，这一章就是为你而准备的。掌握 Express 及其实用性需要对 Node 有基本的认识。如果你用 Node 开发过 Web 程序，则可以跳过本章。在本章中，我们会用 Node 构建一个非常小的 Web 服务器，然后在下一章中介绍如何用 Express 完成相同的任务。

2.1　获取Node

在系统上安装 Node 非常简单。Node 团队做了很多努力，以确保在所有主流平台上都能简单直接地安装 Node。

安装过程非常简单，实际上，它可以总结为以下三个简单的步骤：

(1) 进入 Node 的首页（http://nodejs.org）。
(2) 点击写着“INSTALL”的绿色大按钮。
(3) 按照指令安装。

在 Windows 和 OS X 上，会下载一个安装器，引导你完成整个安装过程。在 Linux 上，如果你用了包管理器（https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager），可能会更快地完成安装并运行。

如果你是 Linux 用户，并且要用包管理器，一定要遵循之前提到的网页上的指令。如果你不加上恰当的包存储库，很多 Linux 发行版都会安装一个非常古老的 Node 版本。

你也可以下载一个独立的安装器（http://nodejs.org/download/），在你向组织内部分发 Node 时会有帮助。

如果你在构建 Node 时遇到困难，或者因为某些原因想从头开始构建 Node，请参考官方安装指南（http://www.joyent.com/blog/installing-node-and-npm/）。

2.2 使用终端

我痴迷于终端（也叫“控制台”或“命令行”）的强大和高效。本书的所有例子都假定你已使用终端。如果你不熟悉你的终端，我强烈建议你花些时间去熟悉它。本书中的很多工具都有 GUI 界面，所以如果你确实不想使用终端，你有自己的选择权，但你就只能靠自己去学习了。

如果你用的是 OS X 或 Linux，有大量历史悠久的 shell（终端命令解释器）可供选择。尽管 zsh 也有它自己的追随者，但目前最流行的还是 bash。我之所以被 bash 吸引，（除了接触时间长之外）主要是因为它的普遍性。在基于 Unix 的机器上，默认的 shell 有 99% 的可能是 bash。

如果你是 Windows 用户，事情就没有那么美好了。微软从不注重在终端上提供令人愉悦的体验，所以你只能多做点工作。Git 中包含一个“Git bash”shell，提供了类似于 Unix 的终端体验（它只有常见 Unix 命令行工具的一个子集，但这个子集很实用）。尽管 Git bash 提供了一个精简的 bash shell，但它用的仍然是内置的 Windows 控制台程序，因此用起来也比较费力（即便像重置控制台窗口大小、选择文本、剪切和粘帖这些简单的功能都是不直观和笨拙的）。因此我推荐你安装 Console2（http://sourceforge.net/projects/console/）或 ConEmu（https://github.com/Maximus5/ConEmu）这些更精致的控制台。对于 Windows 的超级用户，特别是 Windows 系统的 .NET 开发人员，或者骨灰级 Windows 系统和网络的管理员，还有另外一个选择：微软自己的 PowerShell。PowerShell 名符其实，人们可以用它做出非凡的事情，并且技艺娴熟的 PowerShell 用户跟 Unix 命令行大师旗鼓相当。然而，如果你要在 OS X/Linux 和 Windows 之间切换，出于一致性上的考虑，我建议你还是用 Git bash 吧。

Windows 用户还有一种选择：虚拟化。因为现代计算机的架构和能力，虚拟机（VM）的性能实际上已经足以媲美真正的机器了。我们非常幸运能有 Oracle 的免费 VirtualBox，并且 Windows 8 内置了对 VM 的支持。另外，有了像 Dropbox 这样基于云的文件存储，并且 VM 存储和主机存储之间的桥接也很容易，虚拟化更加有吸引力了。与其用 Git bash 给 Windows 羸弱的控制台支持打补丁，还不如用 Linux VM 做开发。如果你觉得 UI 不像你想象的那么平滑，可以使用像 PuTTY（http://www.putty.org/）这样的终端程序，我经常这么做。

最后，不管你用什么系统，都可以使用优秀的 Codio（https://codio.com/）。Codio 是个网站，它可以为你的每个项目起一个新的 Linux 实例，还可以提供一个 IDE 和命令行，并且 Node 也已经安装完毕。它真的非常好用，是快速进入 Node 的极佳方式。

> 如果你在安装 npm 包时指定 `-g`（全局）选项，它们会被装在你的 Windows 主目录的一个子目录下。我发现如果你的用户名中有空格（我的用户名过去是“Ethan Brown”，现在是“ethan.brown”），很多包都会出现问题。出于安全考虑，我建议你选一个没有空格的 Windows 用户名。如果你已经用带有空格的用户名了，建议你创建一个新用户，然后将你的文件传给新账号。重命名你的 Windows 主目录也不是不可能，但充满了危险。

一旦你选定了自己喜欢的 shell，建议你花些时间熟悉一下与它相关的基础知识。网上有很多精彩的教程，你现在应该学习一下，毕竟磨刀不误砍柴工。至少你应该知道如何切换目录，如何复制、移动和删除文件，以及如何中断一个命令行程序（通常是 Ctrl-C）。如果你想变成终端高手，我建议你学一学如何在文件中搜索文本，如何搜索文件和目录，如何把命令链在一起（老式的“Unix 理念”），以及如何重定向输出。

> 在很多类 Unix 的系统上，Ctrl-S 都有特殊的含义：它会“冻结”终端（它曾经被用来暂停快速滚动）。因为“保存”一般也是用这个快捷键，所以经常会有人不假思索地按下这个快捷键，结果大多数人都会被搞糊涂（我也经常犯这个错误）。解冻终端是用 Ctrl-Q，所以如果你忽然发觉终端看起来被冻结了，试一下 Ctrl-Q，看能不能释放它。

2.3 编辑器

很少有话题能像选择编辑器一样在程序员中引起热烈的讨论，其中缘由便是：编辑器是最主要的工具。我用的编辑器是 vi[1]（或者带 vi 模式的编辑器）。并非所有人都喜欢使用 vi（当我告诉同事用 vi 多么容易实现他们在做的事情时，总是会招致他们的白眼），但找一款强大的编辑器并学会如何使用它无疑会极大地提高你的生产率，并且你会享受到个中趣味。我特别喜欢 vi 的原因之一（尽管谈不上是最重要的原因）是它跟 bash 一样，也是普遍存在的。只要你访问 Unix 系统（包括 Cygwin），就能找到 vi。很多流行的编辑器（即便是微软的 Visual Studio！）都有 vi 模式。一旦你习惯了 vi，很难想象还会用其他的编辑器。刚开始接触 vi 时会觉得比较难，但回报是很可观的。

如果你像我一样，了解使用一个普遍存在的编辑器的重要性，也可以选择 Emacs。我对

注 1：近来，vi 与 vim 基本上是同义语。在大部分系统里，vi 成为了 vim 的别名，但我经常键入 vim 来明确我使用的是 vim。

Emacs 一直都不太习惯（通常大多数人选择 Emacs 或者 vi），但我绝对承认 Emacs 的强大和灵活性。如果 vi 的模态编辑方式不适合你，我建议你了解一下 Emacs。

尽管知道控制台编辑器（比如 vi 或 Emacs）可以变得极其方便顺手，你或许还是想要一个更现代化的编辑器。我一些做前端的同事喜欢 Coda，我相信他们的选择。可惜 Coda 只能用在 OS X 上。Sublime Text 是一个强大的现代化编辑器，也有出色的 vi 模式，并且在 Windows、Linux 和 OS X 上都能使用。

Windows 上还有一些很好的免费选择。TextPad 和 Notepad++ 都有它们的支持者。它们都是很强的编辑器，并且你无法抗拒它们的价格诱惑。如果你是 Windows 用户，不要忽视将 Visual Studio 作为 JavaScript 编辑器：它非常地强大，并且它的 JavaScript 自动补足引擎可以称得上是最好的。你可以在微软的官网上免费下载 Visual Studio Express。

2.4 npm

npm 是随处可见的 Node 开发包管理器（我们就是用它获取并安装 Express 的）。"npm"跟 PHP、GNU、WINE 等那些古怪的传统名字不一样，它不是首字母缩写（所以也没有大写），而是"npm 不是缩写"的递归缩写。

从广义上来说，包管理器的两个主要职责是安装开发包和管理依赖项。npm 是一个快速、高能并且毫不费力的包管理器，在 Node 生态系统的高速成长和多样化过程中发挥了重要作用。

当你安装 Node 时就把 npm 装上了，所以如果你是按照前面列出来的步骤安装的 Node，你已经有 npm 了。那么我们开始工作吧！

在使用 npm 时，（毫无悬念）最主要的命令是 `install`。比如要安装 Grunt（一个流行的 JavaScript 任务执行器），你将会（在控制台里）发起下面这个命令：

```
npm install -g grunt-cli
```

标记 `-g` 的意思是告诉 npm 这个包要*全局安装*，即系统全局都可以访问它。在我们讨论 package.json 文件时，这种区别会更明显。就目前而言，JavaScript 工具（比如 Grunt）一般是全局安装的，但你的 Web 程序或项目专用的开发包则不是。

> 不像 Python 语言——从 2.0 升级到 3.0 发生了重大变化，有必要提供一种在不同环境中切换的办法——Node 平台太新了，你很可能总是用最新版的 Node。然而，如果你发现自己确实需要支持多个版本的 Node，有个 nvm（https://github.com/creationix/nvm）项目，可以用它切换环境。

2.5 用Node实现的简单Web服务器

如果你之前曾经做过静态的 HTML 网站，或者有 PHP 或 ASP 背景，可能习惯用 Web 服务器（比如 Apache 或 IIS）提供静态文件服务，以便使用浏览器通过网络查看这些文件。比如说，如果你创建了一个名为 about.html 的文件，并把它放到了恰当的目录下，然后就可以访问 http://localhost/about.html 查看这个文件。根据 Web 服务器的配置，你甚至可以省略 .html，但 URL 和文件名之间的关系很清晰：Web 服务器知道文件在机器的哪个地方，并能把它返回给浏览器。

> 从 localhost 的名字就能看出来，它指的是你所在的机器。这是 IPv4 回环地址 127.0.0.1 或者 IPv6 回环地址 ::1 的常用别名。你应该更常见到 127.0.0.1，不过本书中用的是 localhost。如果你用的是远程的机器（比如通过 SSH 访问的），记得浏览 localhost 时访问的不是你眼前的那台机器。

Node 所提供的范式跟传统的 Web 服务器不同：你写的程序就是 Web 服务器。Node 只是给你提供了一个构建 Web 服务器的框架。

你可能会说“但我不想写 Web 服务器”。这是很自然的反应：你想写一个程序，而不是 Web 服务器。然而在 Node 里编写 Web 服务器非常简单（甚至只需要几行代码），并且你因此取得了对程序的控制权，这是非常值得的。

那么我们开始吧。如果你已经安装了 Node，也已经熟悉了终端，现在一切都准备好了。

2.5.1 Hello World

我发现正规的编程入门范例总是输出毫无创意的“Hello World”消息。但打破这样的传统似乎是不敬之举，所以我们也从这里开始吧，然后再去做一些更有趣的事情。

用你喜欢的编辑器创建一个 helloWorld.js 文件：

```
var http = require('http');

http.createServer(function(req,res){
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('Hello world!');
}).listen(3000);

console.log('Server started on localhost:3000; press Ctrl-C to terminate....');
```

确保是和 helloWorld.js 在同一个目录下，输入 `node hello World.js`。然后打开浏览器访问 http://localhost:3000，你的第一个 Web 服务器就建成啦！这个服务器并没有返回 HTML，而只是向你的浏览器传递了一条普通的文本消息“Hello world!”。如果你想要尝试发送

HTML，可以试验一下：只要把 `text/plain` 换成 `text/html`，再把 `'Hello world!'` 换成一个包含有效 HTML 的字符串就行了。在这里就不演示了，因为我要尽量避免在 JavaScript 里写 HTML，至于原因，我们会在第 7 章深入探讨。

2.5.2 事件驱动编程

Node 的核心理念是*事件驱动编程*。这对程序员来说，意味着你必须知道有哪些事件，以及如何响应这些事件。很多人接触事件驱动编程是从用户界面开始的：用户点击了什么，然后你处理“点击事件”。这个类比很好，因为程序员不能控制用户什么时间点击或者是否会点击，所以事件驱动编程真的很直观。在服务器上响应事件这种概念性的跳跃可能会比较难，但原理是一样的。

在前面那个例子中，事件是隐含的：HTTP 请求就是要处理的事件。`http.createServer` 方法将函数作为一个参数，每次有 HTTP 请求发送过来就会调用那个函数。我们这个简单的程序只是把内容类型设为普通文本，并发送字符串“Hello world!”。

2.5.3 路由

路由是指向客户端提供它所发出的请求内容的机制。对基于 Web 的客户端 / 服务器端程序而言，客户端在 URL 中指明它想要的内容，具体来说就是路径和查询字符串（第 6 章会详细讲解 URL 的组成部分）。

我们扩展一下“Hello world!”那个例子，做些更有意思的事情。做一个有首页、关于页面和未找到页面的极其简单的网站。目前我们还像之前那个例子一样，不提供 HTML，只提供普通文本：

```
var http = require('http');

http.createServer(function(req,res){
        // 规范化 url，去掉查询字符串、可选的反斜杠，并把它变成小写
        var path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase();
        switch(path) {
                case '':
                        res.writeHead(200, { 'Content-Type': 'text/plain' });
                        res.end('Homepage');
                        break;
                case '/about':
                        res.writeHead(200, { 'Content-Type': 'text/plain' });
                        res.end('About');
                        break;
                default:
                        res.writeHead(404, { 'Content-Type': 'text/plain' });
                        res.end('Not Found');
                        break;
        }
```

```
}).listen(3000);

console.log('Server started on localhost:3000; press Ctrl-C to terminate....');
```

运行这段代码，你会发现现在你可以访问首页（http://localhost: 3000）和关于页面（http://localhost:3000/about）。所有查询字符串都会被忽略（所以 http://localhost:3000/?foo=bar 也是返回首页），并且其他所有 URL（http://localhost:3000/foo）返回的都是未找到页面。

2.5.4 静态资源服务

现在我们有了一些可用的简单路由，接下来我们提供一些真正的 HTML 和 logo 图片。因为这些内容不会变化，所以它们都被称为“静态资源”（相对于股票之类的内容，你每次刷新页面，股价都会变化）。

> 用 Node 提供静态资源只适用于初期的小型项目，对于比较大的项目，你应该会想用 Nginx 或 CDN 之类的代理服务器来提供静态资源。对此，第 16 章会有更多介绍。

如果你用过 Apache 或 IIS，可能习惯于只是创建一个 HTML 文件，访问它，然后让它自动发送到客户端。Node 不是那样的：我们必须打开文件，读取其中的内容，然后将这些内容发送给浏览器。所以我们要在项目里创建一个名为 public 的目录（在下一章中，你就会明白我们为什么不管它叫 static）。在这个目录下创建文件 home.html、about.html、notfound.html，子目录 img，以及一个名为 img/logo.jpg 的图片。以上这些工作就由你自己来完成了：既然你在阅读这本书，那么你应该知道怎么编写 HTML 文件和找张图片。在你的 HTML 文件中这样引用 logo：<img href="/img/logo.jpg" alt="logo">。

接下来修改 helloWorld.js：

```
var http = require('http'),
        fs = require('fs');

function serveStaticFile(res, path, contentType, responseCode) {
        if(!responseCode) responseCode = 200;
        fs.readFile(__dirname + path, function(err,data) {
                if(err) {
                        res.writeHead(500, { 'Content-Type': 'text/plain' });
                        res.end('500 - Internal Error');
                } else {
                        res.writeHead(responseCode,
                                { 'Content-Type': contentType });
                        res.end(data);
                }
        });
}
```

```
http.createServer(function(req,res){
        // 规范化 url，去掉查询字符串、可选的反斜杠，并把它变成小写
        var path = req.url.replace(/\/?(?:\?.*)?$/, '')
                .toLowerCase();
        switch(path) {
                case '':
                        serveStaticFile(res, '/public/home.html', 'text/html');
                        break;
                case '/about':
                        serveStaticFile(res, '/public/about.html', 'text/html');
                        break;
                case '/img/logo.jpg':
                        serveStaticFile(res, '/public/img/logo.jpg',
                                'image/jpeg');
                        break;
                default:
                        serveStaticFile(res, '/public/404.html', 'text/html',
                                404);
                        break;
        }
}).listen(3000);

console.log('Server started on localhost:3000; press Ctrl-C to terminate....');
```

这个例子中，我们的路由是非常缺乏想象力的。如果你访问 http://localhost:3000/about，就返回 public/about.html 文件。你可以随意修改路由，也可以随意修改文件。比如说，如果你一周里的每一天都要换一个关于页面，你可能会有 public/about_mon.html、public/about_tue.html 等之类的页面，在你的路由中定义好逻辑，从而在用户访问 http://localhost:3000/about 时能提供恰当的页面。

注意，我们创建了一个辅助函数 `serveStaticFile`，它完成了大部分工作。`fs.readFile` 是读取文件的异步方法。这个函数有同步版本，`fs.readFileSync`，但这种异步思考问题的方式，你接触得越早越好。这个函数不复杂：它调用 `fs.readFile` 读取指定文件中的内容。`fs.readFile` 读取完文件后执行回调函数，如果文件不存在，或者读取文件时遇到许可权限方面的问题，会设定 `err` 变量，并且会返回一个 HTTP 500 的状态码表明服务器错误。如果文件读取成功，文件会带着特定的响应码和内容类型发给客户端。第 6 章还会详细讨论响应码。

`__dirname` 会被解析为正在执行的脚本所在的目录。所以如果你的脚本放在 `/home/sites/app.js` 中，则 `__dirname` 会被解析为 `/home/sites`。不管什么时候，这个全局变量用起来都很方便。如果不这么做，在不同的目录中运行你的程序时很可能会出现难以诊断的错误。

2.6 走向Express

到目前为止，Node 貌似没什么能打动你的地方。我们基本上是在重复 Apache 或 IIS 可自动完成的工作，但现在你已经了解了 Node 是如何工作的，也知道你拥有多少控制权。我们还没做出特别值得称道的事情，但可预见到我们可将它作为跳板去完成更加复杂的事情。如果我们沿着这条路走下去，写出越来越复杂的 Node 程序，最后你可能会得到一个类似于 Express 的东西……

幸运的是，我们没必要这样做，因为 Express 已经存在了，你不用自己花那么多时间去写基础设施类的代码。既然现在已经掌握了一点 Node 方面的知识，那么让我们准备学习 Express 吧。

第3章

省时省力的Express

第 2 章介绍了如何用 Node 创建一个简单的 Web 服务器，本章会用 Express 再次创建该服务器。本章是本书后续内容的起点，会介绍 Express 的基础内容。

3.1 脚手架

脚手架并不是一个新想法，但很多人（包括我自己）都是通过 Ruby 才接触到这个概念的。这个想法很简单：大多数项目都需要一定数量的“套路化”代码，谁会想每次开始新项目时都重新写一次这些代码呢？对此有个简单的方法，那就是创建一个通用的项目骨架，每次开始新项目时，只需复制这个骨架，或者说是模板。

RoR 把这个概念向前推进了一步，它提供了一个可以自动生成脚手架的程序。相对于从一堆模板中作出选择，这种方式的优点是可以生成更复杂的框架。

Express 借鉴了 RoR 的这一做法，提供了一个生成脚手架的工具，从而可以让你开始一个新的 Express 项目。

尽管 Express 有可用的脚手架工具，但它目前并不能生成我在本书中推荐使用的框架。特别是它不支持我所选择的模板语言（Handlebars），也没有遵循我所偏好的命名规则（尽管这很容易解决）。

尽管我们不用这个脚手架工具，但我还是建议你在读完本书后看一下它：到那时，你就能够充分了解它生成的脚手架是否对你有用了。

套路化对最终发送到客户端的真正 HTML 也是有用的。我推荐非常出色的 HTML5

Boilerplate（http://html5boilerplate.com/），它能生成一个很不错的空白 HTML5 网站。最近 HTML5 Boilerplate 又新增加了可定制的功能，其中一个定制选项包含 Twitter Bootstrap，这个是我高度推荐的前端框架。在第 7 章，我们会用一个基于 Bootstrap 的版本创建一个响应式的现代化 HTML5 网站。

3.2　草地鹨旅行社网站

本书以一个可运行的网站为例：假想的草地鹨旅行社网站，该旅行社是一家为到俄勒冈州旅游的人提供服务的公司。如果你对创建 REST 应用程序更感兴趣，不用担心，因为草地鹨旅行社网站除了作为功能性网站外，也提供 REST 服务。

3.3　初始步骤

先给你的项目创建一个新目录，这将作为项目的根目录。本书中，凡提到“项目目录”“程序目录”或“项目根路径”，指的都是这个目录。

> 或许你会把 Web 程序文件跟项目相关的其他文件全都分开存放，比如会议纪要、文档等。因此，我建议你把项目根路径作为项目目录的子目录。比如，对于草地鹨旅行社网站而言，我会把项目放在 ~/projects/meadowlark，而项目根路径放在 ~/projects/meadowlark/site。

npm 在 package.json 文件中管理项目的依赖项以及项目的元数据。要创建这个文件，最简单的办法是运行 `npm init`：它会问一系列的问题，然后为你生成一个 package.json 文件帮你起步（对于“入口点”的问题，用 meadowlark.js 或项目的名字作为答案）。

> 如果你的 package.json 文件中没有指定一个存储库的 URL，以及一个非空的 README.md 文件，那么你每次运行 npm 时都会看到警告信息。package.json 文件中的元数据只有在发布到 npm 存储库时才是真正必要的，但为了消除 npm 的警告信息，做这些工作依然是值得的。

第一步是安装 Express。运行下面这条 npm 命令：

```
npm install --save express
```

运行 `npm install` 会把指定名称的包安装到 node_modules 目录下。如果你用了 `--save` 选项，它还会更新 package.json 文件。因为 node_modules 随时都可以用 npm 重新生成，所以我们不会把这个目录保存在我们的代码库中。为了确保不把它添加到代码库中，我们可以创建一个 .gitignore 文件：

```
# ignore packages installed by npm
node_modules

# put any other files you don't want to check in here,
# such as .DS_Store (OSX), *.bak, etc.
```

接下来创建 meadowlark.js 文件，这是我们项目的入口。本书中将这个文件简单称为“程序文件”：

```
var express = require('express');

var app = express();

app.set('port', process.env.PORT || 3000);

// 定制 404 页面
app.use(function(req, res){
        res.type('text/plain');
        res.status(404);
        res.send('404 - Not Found');
});

// 定制 500 页面
app.use(function(err, req, res, next){
        console.error(err.stack);
        res.type('text/plain');
        res.status(500);
        res.send('500 - Server Error');
});

app.listen(app.get('port'), function(){
  console.log( 'Express started on http://localhost:' +
    app.get('port') + '; press Ctrl-C to terminate.' );
});
```

很多教程，甚至是 Express 的脚手架生成器会建议你把主文件命名为 app.js（或者有时是 index.js 或 server.js）。除非你用的托管服务或部署系统对程序主文件的名称有特定的要求，否则我认为这么做是没有道理的，我更倾向于按照项目命名主文件。凡是曾在编辑器里见过一堆 index.html 标签的人都会立刻明白这样做的好处。`npm init` 默认是用 index.js，如果要使用其他的主文件名，要记得修改 package.json 文件中的 `main` 属性。

现在你有了一个非常精简的 Express 服务器。你可以启动这个服务器（node meadowlark.js），然后访问 http://localhost:3000。结果可能会让你失望，因为你还没给 Express 任何路由信息，所以它会返回一个 404 页面，表示你访问的页面不存在。

注意我们指定程序端口的方式：`app.set(port, process.env.PORT || 3000)`。这样我们可以在启动服务器前通过设置环境变量覆盖端口。如果你在运行这个案例时发现它监听的不是 3000 端口，检查一下是否设置了环境变量 `PORT`。

我高度推荐你安装一个能显示 HTTP 请求状态码和所有重定向的浏览器插件。这样在解决重定向问题或者不正确的状态码时会更加容易，它们经常被忽视。对于 Chrome 来说，Ayima 的 Redirect Path 特别好用。在大多数浏览器中，都能在开发者工具的网络部分看到状态码。

我们来给首页和关于页面加上路由。在 404 处理器之前加上两个新路由：

```
app.get('/', function(req, res){
        res.type('text/plain');
        res.send('Meadowlark Travel');
});
app.get('/about', function(req, res){
        res.type('text/plain');
        res.send('About Meadowlark Travel');
});

// 定制 404 页面
app.use(function(req, res, next){
       res.type('text/plain');
       res.status(404);
       res.send('404 - Not Found');
});
```

`app.get` 是我们添加路由的方法。在 Express 文档中写的是 `app.VERB`。这并不意味着存在一个叫 `VERB` 的方法，它是用来指代 HTTP 动词的（最常见的是“get”和“post”）。这个方法有两个参数：一个路径和一个函数。

路由就是由这个路径定义的。`app.VERB` 帮我们做了很多工作：它默认忽略了大小写或反斜杠，并且在进行匹配时也不考虑查询字符串。所以针对关于页面的路由对于 /about、/About、/about/、/about?foo=bar、/about/?foo=bar 等路径都适用。

路由匹配上之后就会调用你提供的函数，并把请求和响应对象作为参数传给这个函数，我们在第 6 章会详细介绍这两个对象。现在我们只是返回了状态码为 200 的普通文本（Express 默认的状态码是 200，不用显式指定）。

我们这次使用的不是 Node 的 `res.end`，而是换成了 Express 的扩展 `res.send`。我们还用 `res.set` 和 `res.status` 替换了 Node 的 `res.writeHead`。Express 还提供了一个 `res.type` 方法，可以方便地设置响应头 `Content-Type`。尽管仍然可以使用 `res.writeHead` 和 `res.end`，但没有必要也不作推荐。

注意，我们对定制的 404 和 500 页面的处理与对普通页面的处理应有所区别：用的不是 `app.get`，而是 `app.use`。`app.use` 是 Express 添加中间件的一种方法。我们会在第 10 章更深入地探讨中间件，现在你可以把它看作处理所有没有路由匹配路径的处理器。这里涉及一个非常重要的知识点：在 Express 中，路由和中间件的添加顺序至关重要。如果我们把 404 处理器放在所有路由上面，那首页和关于页面就不能用了，访问这些 URL 得到的都是 404。现在我们的路由相当简单，但其实它们还能支持通配符，这会导致顺序上的问题。比如说，如果要给关于页面添加子页面，比如 /about/contact 和 /about/directions 会怎么样呢？下面这段代码是达不到预期效果的：

```
app.get('/about*',function(req,res){
        // 发送内容……
})
app.get('/about/contact',function(req,res){
        // 发送内容……
})
app.get('/about/directions',function(req,res){
        // 发送内容……
})
```

本例中的 `/about/contact` 和 `/about/directions` 处理器永远无法匹配到这些路径，因为第一个处理器的路径中用了通配符：`/about*`。

Express 能根据回调函数中参数的个数区分 404 和 500 处理器。第 10 章和 12 章会详细介绍错误路由。

你可以再次启动服务器，现在首页和关于页面都可以运行了。

截至目前我们所做的事情，即使不用 Express 也很容易完成，但 Express 所提供的一些功能并非那么显而易见。还记得上一章我们是如何规范化 `req.url` 来确定所请求的资源吗？我们必须手动剥离查询字符串和反斜杠，并转化为小写。而 Express 的路由器会自动帮我们处理好这些细节。尽管目前看起来这并非什么大不了的事情，但这只是 Express 路由器能力的冰山一角。

3.3.1 视图和布局

如果你熟知“模型 – 视图 – 控制器”模式，那你对视图这个概念应该不会感到陌生。视图本质上是要发送给用户的东西。对网站而言，视图通常就是 HTML，尽管也会发送 PNG 或 PDF，或者其他任何能被客户端渲染的东西。不过，本书中的视图是指 HTML。

视图与静态资源（比如图片或 CSS 文件）的区别是它不一定是静态的：HTML 可以动态构建，为每个请求提供定制的页面。

Express 支持多种不同的视图引擎，它们有不同层次的抽象。Express 比较偏好的视图引擎是 Jade（因为它也是 TJ Holowaychuk 开发的）。Jade 所采用的方式非常精简：你写的根本不像是 HTML，因为没有尖括号和结束标签，这样可以少敲好多次键盘。然后，Jade 引擎会将其转换成 HTML。

Jade 是非常吸引人的，但这种程度的抽象也是有代价的。如果你是一名前端开发人员，即便你实际上是用 Jade 编写视图，也必须理解 HTML，并且有足够深入的认识。我认识的大多数前端开发人员都不喜欢他们主要的标记语言被抽象化处理。因此我推荐使用另外一个抽象程度较低的模板框架 Handlebars。Handlebars（基于与语言无关的流行模板语言 Mustache）不会试图对 HTML 进行抽象：你编写的是带特殊标签的 HTML，Handlebars 可以借此插入内容。

为了支持 Handlebars，我们要用到 Eric Ferraiuolo 的 `express3-handlebars` 包（尽管名字中是 express3，但这个包在 Express 4.0 中也可以使用）。在你的项目目录下执行：

```
npm install --save express3-handlebars
```

然后在创建 app 之后，把下面的代码加到 meadowlark.js 中：

```
var app = express();

// 设置 handlebars 视图引擎
var handlebars = require('express3-handlebars')
        .create({ defaultLayout:'main' });
app.engine('handlebars', handlebars.engine);
app.set('view engine', 'handlebars');
```

这段代码创建了一个视图引擎，并对 Express 进行了配置，将其作为默认的视图引擎。接下来创建 views 目录，在其中创建一个子目录 layouts。如果你是一位经验丰富的 Web 开发人员，可能已经熟悉布局的概念了（有时也被称为“母版页”）。在开发网站时，每个页面上肯定有一定数量的 HTML 是相同的，或者非常相近。在每个页面上重复写这些代码不仅非常繁琐，还会导致潜在的维护困境：如果你想在每个页面上做一些修改，那就要修改所有文件。布局可以解决这个问题，它为网站上的所有页面提供了一个通用的框架。

所以我们要给网站创建一个模板。接下来我们创建一个 views/layouts/main.handlebars 文件：

```
<!doctype html>
<html>
<head>
    <title>Meadowlark Travel</title>
</head>
<body>
    {{{body}}}
</body>
</html>
```

以上内容你未曾见过的可能只有 {{{body}}}。这个表达式会被每个视图自己的 HTML 取代。在创建 Handlebars 实例时，我们指明了默认布局（defaultLayout:'main'）。这就意味着除非你特别指明，否则所有视图用的都是这个布局。

接下来我们给首页创建视图页面，views/home.handlebars：

```
<h1>Welcome to Meadowlark Travel</h1>
```

关于页面，views/about.handlebars：

```
<h1>About Meadowlark Travel</h1>
```

未找到页面，views/404.handlebars：

```
<h1>404 - Not Found</h1>
```

最后是服务器错误页面，views/500.handlebars：

```
<h1>500 - Server Error</h1>
```

你或许想在编辑器中把 .handlebars 和 .hbs（另外一种常见的 Handlebars 文件扩展名）跟 HTML 相关联，以便启用语法高亮和其他编辑器特性。如果是 vim，你可以在 ~/.vimrc 文件中加上一行 au BufNewFile,BufRead *.handlebars set file type=html。其他编辑器请参考相关文档。

现在视图已经设置好了，接下来我们必须将使用这些视图的新路由替换旧路由：

```
app.get('/', function(req, res) {
        res.render('home');
});
app.get('/about', function(req, res) {
        res.render('about');
});

// 404 catch-all 处理器（中间件）
app.use(function(req, res, next){
       res.status(404);
       res.render('404');
});

// 500 错误处理器（中间件）
app.use(function(err, req, res, next){
       console.error(err.stack);
       res.status(500);
       res.render('500');
});
```

需要注意，我们已经不再指定内容类型和状态码了：视图引擎默认会返回 text/html 的内

容类型和 200 的状态码。在 catch-all 处理器（提供定制的 404 页面）以及 500 处理器中，我们必须明确设定状态码。

如果你再次启动服务器检查首页和关于页面，将会看到那些视图已呈现出来。如果你检查源码，将会看到 views/layouts/main.handlebars 中的套路化 HTML。

3.3.2 视图和静态文件

Express 靠中间件处理静态文件和视图。第 10 章会更详细地介绍中间件的概念。现在只需了解中间件是一种模块化手段，它使得请求的处理更加容易。

`static` 中间件可以将一个或多个目录指派为包含静态资源的目录，其中的资源不经过任何特殊处理直接发送到客户端。你可以在其中放图片、CSS 文件、客户端 JavaScript 文件之类的资源。

在项目目录下创建名为 public 的子目录（因为这个目录中的所有文件都会直接对外开放，所以我们称这个目录为 public）。接下来，你应该把 `static` 中间件加在所有路由之前：

```
app.use(express.static(__dirname + '/public'));
```

`static` 中间件相当于给你想要发送的所有静态文件创建了一个路由，渲染文件并发送给客户端。接下来我们在 public 下面创建一个子目录 img，并把 logo.png 文件放在其中。

现在我们可以直接指向 /img/logo.png（注意：路径中没有 public，这个目录对客户端来说是隐形的），`static` 中间件会返回这个文件，并正确设定内容类型。接下来我们修改一下布局文件，以便让我们的 logo 出现在所有页面上：

```
<body>
    <header><img src="/img/logo.png" alt="Meadowlark Travel Logo"></header>
    {{{body}}}
</body>
```

<header> 是 HTML5 中引入的元素，它出现在页面顶部，提供一些与内容有关的额外语义信息，比如 logo、标题文本或导航等。

3.3.3 视图中的动态内容

视图并不只是一种传递静态 HTML 的复杂方式（尽管它们当然能做到）。视图真正的强大之处在于它可以包含动态信息。

比如在关于页面上发送“虚拟幸运饼干”。我们在 meadowlark.js 中定义一个幸运饼干数组：

```
var fortunes = [
        "Conquer your fears or they will conquer you.",
        "Rivers need springs.",
        "Do not fear what you don't know.",
        "You will have a pleasant surprise.",
        "Whenever possible, keep it simple.",
];
```

修改视图（/views/about.handlebars）以显示幸运饼干：

```
<h1>About Meadowlark Travel</h1>

<p>Your fortune for the day:</p>
<blockquote>{{fortune}}</blockquote>
```

接下来修改路由 /about，随机发送幸运饼干：

```
app.get('/about', function(req, res){
        var randomFortune =
                fortunes[Math.floor(Math.random() * fortunes.length)];
        res.render('about', { fortune: randomFortune });
});
```

重启服务器，加载 /about 页面，你会看到一个随机发放的幸运饼干。模板真的是非常有用，我们将在第 7 章详细介绍。

3.4 小结

我们刚用 Express 创建了一个非常基本的网站。尽管简单，但这个网站包含了功能完备的网站所需的一切。在下一章中，我们会事无巨细地介绍为增加更高级功能需做的准备工作。

第 4 章

工欲善其事，必先利其器

在前面两章中，我们只是试验了一下，可以说是小试牛刀。在实现更复杂的功能之前，我们先做一些准备工作，并培养一些好的工作习惯。

在本章中，我们将开始我们的草地鹨旅行社项目。然而在开始搭建网站之前，要先确保我们有制作高质量产品所需的工具。

> 你不一定非要按照本书中的例子来做。如果你迫切地想要搭建自己的网站，可以参照这个例子的框架，以此为基础进行相应的修改，这样等你读完本书时，就有一个已经完工的网站了！

4.1 最佳实践

最近你应该听了很多次“最佳实践”这个词，它的意思是你应该“正确地做事”，不要走捷径（我们马上就会讨论它的确切含义）。毫无疑问，你一定听过那句工程格言：面对“快速”“低廉”“优质”三个选项，你总是只能任选其中两个。这个模型总会困扰我，因为它没考虑正确做事的累计价值。你第一次正确做事所用的时间可能是你马马虎虎迅速做事所需时间的 5 倍。然而第二次将只需要 3 倍的时间。等你做过很多次后，正确做事的速度几乎能与马马虎虎迅速做事一样了。

有一位击剑教练总是提醒我们，熟并不能生巧：熟练的能变成永久不变的。也就是说，如果你一次又一次地做同一件事，最终它将变成下意识的、机械式的。确实如此，但它没考虑你不断练习做某件事情时的品质。如果你按照坏习惯练习，坏习惯就变成机械式的了。

所以你应该遵循完美的规则去练习，这样才能成就完美。因此我希望接下来你能遵循本书中的示例，就好像你在搭建一个真实的网站，就好像你的声誉和报酬都取决于这次产出的品质。你不仅要从本书中学习新技能，还要通过练习培养好习惯。

我们练习的重点是版本控制和质量保证。本章会讨论版本控制，下一章将讨论质量保证。

4.2 版本控制

我不必再跟你解释版本控制的价值了吧（它可能需要一整本书）！大体而言，版本控制有以下益处：

- 文档
  能够回溯项目的历史，回顾所做的决策及组件的开发顺序，可形成宝贵的文档。记录项目的历史是十分有价值的。

- 归属
  如果你在一个团队中工作，归属极其重要。当你发现代码模糊不清或有问题时，知道是谁做的修改可以节省你很多时间。也许，与这个修改相关的评论就足以解决你的疑问了，但如果不能，你也知道应该和谁沟通。

- 试验
  一个好的版本控制系统能让你做试验。你可以引出一个分支，尝试做一些新的东西，不用担心会影响项目的稳定性。如果试验成功，你可以把它纳入到项目中；如果不成功，你可以放弃它。

几年前我开始用分布式版本控制系统（DVCS）。我把选择的范围缩小到只有 Git 和 Mercurial，最终因为 Git 的普及程度和灵活性选定了 Git。这两个都是优秀的免费版本控制系统，我建议你选择其中的一个。本书用的是 Git，但你也可以用 Mercurial（或者其他版本控制系统）。

如果你不了解 Git，建议你看一下 Jon Loeliger 的 Version Control with Git (O'Reilly，http://shop.oreilly.com/product/9780596520137.do）。另外，Code School 也有很好的 Git 入门课程（https://try.github.io/）。

4.3 针对本书如何使用Git

首先确保你已经安装了 Git。输入 `git --version`，如果没有输出版本号，那你还需要安装一下 Git。请参见 Git 文档（http://git-scm.com/）中的安装指南。

参照本书中的例子有两种方式。一种是自己录入示例，并参照其中的 Git 命令。另一种是

克隆我给所有示例用的 Git 存储库，并检出每个例子的相关标签。有些人自己录入示例可以学得更好，而有些人则偏好只是观察，然后做些修改，而不是全部录入。

4.3.1 如果你要自己动手

我们的项目已经有了一个非常粗略的框架：一些视图，一个布局，一个 logo，一个主程序文件，一个 package.json 文件。接下来我们继续推进，创建一个 Git 存储库并加入所有文件。

首先，我们进入项目目录并创建一个 Git 存储库：

```
git init
```

在添加这些文件之前，需要创建一个 .gitignore 文件，以防不慎把不想添加的东西加进去。在项目目录中创建一个文本文件 .gitignore，你可以把任何想让 Git 默认忽略的文件或目录写在该文件里（每个一行）。它还支持通配符。比如说，如果你的编辑器会创建带波形号的备份文件（比如 meadowlark.js~），你可以在 .gitignore 文件中放入 `*~`。如果你用的是 Mac，应该还要在这个文件里加入 `.DS_Store`，还有 `node_modules`（马上讲述原因）。所以这个文件看起来可能是这样的：

```
node_modules
*~
.DS_Store
```

> .gitignore 文件中的条目也适用于子目录。所以如果你把 `*~` 放在项目根目录下的 .gitignore 文件里，那么子目录里的所有这种备份文件都会被忽略。

现在我们可以把所有已有的文件都加到 Git 里，这有很多种做法。我一般喜欢用 `git add -A`，这是所有方法中最彻底的。如果你刚接触 Git，且只想提交一两个文件，我建议你逐一添加文件（比如 `git add meadowlark.js`）；如果你想添加所有的修改（包括对文件的删除操作），则用 `git add -A`。因为我们想添加做过的所有工作，所以使用：

```
git add -A
```

> 新手一般都会对 `git add` 命令感到困惑：它添加的是修改，而不是文件。所以，如果你修改过 meadowlark.js，然后输入 `git add meadowlark.js`，你真正所做的是把刚刚做过的修改添加了进来。

Git 有一个“暂存区”，当你执行 `git add` 时，这些修改就被存放在该区域中。所以我们刚才添加的修改实际上还没提交，但它们已经准备就绪了。要提交这些修改，用 `git commit`：

```
git commit -m "Initial commit."
```

`-m "Initial commit."` 是写一条与这次提交相关的消息。Git 甚至不允许没有消息的提交，这种要求是很有道理的。无论何时，一定要尽量提供有意义的提交消息，它们应该简明扼要地描述你所做的工作。

4.3.2　如果你要使用官方存储库

对于官方存储库，每次添加或修改已有源码我都会创建一个标签。要用它作为起点，只要克隆一下：

```
git clone https://github.com/EthanRBrown/web-development-with-node-and-express
```

为了方便，在每一章的开始部分我都添加了一个标签（一般指向前一章的最后一次提交）。所以现在你只要检出跟本章关联的标签就可以了：

```
git checkout ch04
```

注意，章节标签（比如 ch04）表示你即将进入那一章时项目的状态，是在讨论任何内容之前，有时还可能伴随着前一章的最后一个标签。随着章节向前推进，在讨论完其中内容之后还会添加标签。比如，当你看到后面的“npm 包”这一节时，可以检出标签为 ch04-npm-packages 的源码，查看在这一节中讨论的变化。并不是每一节都有对应的标签，但我会尽量确保存储库易于理解。了解更多有关存储库如何组织的信息，请参见 README 文件。

> 如果你到某一点时想要做试验，记得你检出的标签要将你置于一种 Git 称为“分离的 HEAD”的状态中。尽管你可以随意编辑任何文件，但如果你不先创建一个分支，提交任何修改都是不安全的。所以如果你确实想要基于一个标签做一个试验性的分支，只需创建一个新分支后检出，只要一个命令就可以做到：`git checkout -b experiment`（experiment 是分支的名字，你可以用你喜欢的任何名字）。然后，你就可以安全地在这个分支上随意编辑和提交了。

4.4　npm包

项目所依赖的 npm 包放在 node_modules 目录下（很遗憾这个包的名字为 node_modules 而不是 npm_packages，因为 Node 模块是一个相关但不同的概念）。如果你想满足自己的好奇

心，抑或是为了调试程序，可以随意浏览这个目录，但永远不要修改这个目录中的任何代码。这是因为那不仅是不良的行为，而且你所做的修改很可能轻易地就被 npm 消除了。如果你想对项目所依赖的包进行修改，正确的做法应该是创建那个项目的副本。如果你确实采取了这种策略，并且觉得自己的改进也能帮到其他人，恭喜你，你现在已经参与到一个开源项目中来了！你可以提交自己的修改，如果这些修改符合项目的标准，它们会被纳入到官方包中。改善已有包和创建定制包超出了本书的范围，但如果你想改善已有包，可以向活跃的开发者社区寻求帮助。

package.json 文件有双重作用：描述项目和列出依赖项。现在去看看你的 package.json 文件，你应该能看到：

```
{
  "dependencies": {
    "express": "^4.0.0",
    "express3-handlebars": "^0.5.0"
  }
}
```

现在我们的 package.json 文件里只有与依赖项相关的信息。注意包版本号之前的插入符（^），这表明在下一个主要版本号之前，所有以指定版本号开始的版本都能用。比如说，这个 package.json 中的 Express，从 4.0.0 开始都能用，所以 4.0.1 和 4.9.9 都可以，但 3.4.7 不行，5.0.0 也不行。这是使用 `npm install --save` 时默认指定的版本范围，并且通常也很安全。这种方式的结果是如果你想升级到新版本，就只能编辑这个文件来指定新版本。一般来说，这是好事，因为这样可以防止依赖项的变化在你不知情的情况下破坏项目。npm 中的版本号是由组件 semver（表示"语义版本器"）解析的。如果你想了解 npm 中更多与版本有关的信息，可以翻阅一下 semver 的文档（https://www.npmjs.org/doc/misc/semver.html）。

因为 package.json 文件中列出了所有的依赖项，所以说 node_modules 目录实际上是个衍生品。这就是说，如果你把它删了，要让项目重新恢复工作，只需运行 `npm install`，它便会重建这个目录，并把所有必需的依赖项全放进去。因此我建议把 node_modules 放在 .gitignore 文件中，不要把它纳入到源码的版本控制中去。然而也有人觉得存储库中应该包含运行项目所必需的一切东西，他们更愿意把 node_modules 放在源码的版本控制中。我觉得这是存储库中的"噪音"，我更偏向于忽略它。

不管什么时候在项目中使用了 Node 模块，你都要确保它作为依赖项出现在 package.json 文件中。如果你没能做到这一点，npm 将无法构建出恰当的依赖项，而当其他开发人员检出项目时（或者当你换了一台机器时），就无法安装正确的依赖项，包管理器的价值也不能得到有效发挥。

4.5 项目元数据

package.json 文件的另一个作用便是存放项目的元数据，比如项目名称、作者、授权信息等。如果你用 `npm init` 来初始化创建 package.json 文件，它会为你生成必需的域，然后你可以随时修改它们。如果你想把项目放到 npm 或 Github 上，则对元数据的要求会比较严格。如果你想了解更多有关 package.json 中各个域的信息，请查阅 package.json 的文档（https://www.npmjs.org/doc/files/package.json.html）。另一个重要的元数据是 README.md 文件。这个文件很适合描述网站的整体架构，也适合于存放刚接触项目的人需要了解的重要信息。这个文件是用基于 Markdown 的文本维基格式写成的。更多信息请查阅 Markdown 文档（http://daringfireball.net/projects/markdown/）。

4.6 Node模块

前面提到过，Node 模块和 npm 包是两个相互关联但又彼此不同的概念。Node 模块，就像它的名字一样，提供了一个模块化和封装的机制。npm 包则提供了一种存储、版本化和引用项目（不限于模块）的标准范式。比如，我们在主程序文件中将 Express 作为一个模块引入：

```
var express = require('express');
```

`require` 是一个用来引入模块的 Node 函数。Node 默认会在目录 node_modules（这应该不足为奇，在 node_modules 目录下有个 express 目录）中寻找这些模块。然而 Node 还提供了创建自有模块的机制（你永远不要在 node_modules 中创建自己的模块）。接下来，我们看看如何将上一章实现的幸运饼干功能模块化。

首先，我们创建一个用来保存模块的目录。名字随意，但一般都称为 lib（library 的缩写）。在这个目录下创建一个 fortune.js 文件：

```
var fortuneCookies = [
        "Conquer your fears or they will conquer you.",
        "Rivers need springs.",
        "Do not fear what you don't know.",
        "You will have a pleasant surprise.",
        "Whenever possible, keep it simple.",
];

exports.getFortune = function() {
        var idx = Math.floor(Math.random() * fortuneCookies.length);
        return fortuneCookies[idx];
};
```

这里要特别注意全局变量输出的用法。如果你想让一个东西在模块外可见，必须把它加到 `exports` 上。在这个例子中，在模块外可以访问到函数 `getFortune`，但数组 `fortuneCookies`

是完全隐藏起来的。这是一件好事，因为封装可以减少容易出错和较脆弱的代码。

> 有几种从模块中输出功能的方法。本书会讲到各种不同的方法，并在第 22 章中进行汇总。

我们现在可以从 meadowlark.js 中移除 `fortuneCookies` 数组（尽管留下它也没什么坏处，因为它绝不会跟 lib/fortune.js 中定义的同名数组产生冲突）。按惯例（但不是必须），在文件的顶部要指明引入什么，所以我们在 meadowlark.js 文件的顶部加上下面这行代码：

```
var fortune = require('./lib/fortune.js');
```

注意，我们在模块名称前加了前缀 `./`。这是告诉 Node，它不应该到 node_modules 目录中查找这个模块，如果我们忽略了这个前缀就会导致失败。

接下来在关于页面的路由中，我们可以利用以上模块里的 `getFortune` 方法：

```
app.get('/about', function(req, res) {
        res.render('about', { fortune: fortune.getFortune() } );
});
```

如果你一直在按照步骤操作，现在可以提交这些修改了：

```
git add -A
git commit -m "Moved 'fortune cookie' functionality into module."
```

或者如果你在用官方存储库，则可以看这个标签中的变化：

```
git checkout ch04
```

你将会发现用模块封装功能既强大又简便，它能改善项目的总体设计和可维护性，还能使测试变得更加容易。了解更多信息，请参考 Node 模块的官方文档（http://nodejs.org/api/modules.html）。

第5章

质量保证

很不幸，质量保证是一个很容易让开发人员感到恐惧的词汇。毕竟，每个人都希望制作出高品质的软件。所以最终目标不是症结，政治问题才是。我发现 Web 开发中一般会出现两种情况：

- 大型或资金充裕的组织

  这些组织通常会有 QA 部门，并且不幸的是，QA 部门和开发部门之间是一种敌对关系。这是最糟糕的事情。两个部门都在相同的团队中，目标一致，但 QA 成功的标准是找到更多 bug，而开发成功的标准一般是产生较少的 bug，因此形成了冲突和竞争的基础。

- 小型组织或预算有限的组织

  这些组织通常没有 QA 部门，开发人员既要开发软件，又要承担 QA 工作。这不是荒谬的想象，或者利益冲突。然而 QA 跟开发大不相同，它需要不同的个性和才能。这并不是不可能的，确实有些开发人员有 QA 的思维模式，但当最终期限临近时，在 QA 上投入的力量往往无法保证，从而对项目造成损害。

大多数现实生活中的工作都需要多种技能，并且渐渐地，个人越来越难成为掌握所有这些技能的专家。然而，具备某些职责之外的技能可以提升你在团队中的地位，也可以使团队的工作更加高效。具备 QA 技能的开发人员就是如此：这两种工作连接得如此紧密，以至于跨学科的理解力变得极有价值。

业界还有一种将 QA 和开发岗位融合的趋势，让开发人员负责 QA。在这种范式下，由擅长 QA 的软件工程师担任开发人员的顾问，帮他们将 QA 植入到开发流程中。不管 QA 岗

位是分散的还是集中的，了解 QA 对开发人员都是有益的。

这本书不是面向 QA 专家的，而是面向开发人员的。所以我的目标不是把你变成 QA 专家，而是介绍一些这方面的经验。如果你所在的组织有专职 QA，与他们沟通和协作将会变得更容易。如果没有，它可以是一个起始点，让你为项目建立一个完备的 QA 方案。

5.1 QA：值得吗

QA 的成本很高，而且有时候非常高。那它到底值不值得呢？这个计算起来很复杂。大多数组织都会使用某种“投入产出”模型。如果你花了钱，那么肯定希望至少能收回成本（多了更好）。然而对于 QA 而言，投入和产出之间的关系很难厘清。比如说，一个完善并广受好评的产品与一个新的、不知名的项目相比，可能要花费更长的时间去处理质量问题。很显然，没有人想生产质量低下的产品，但技术上的压力很大。推向市场的时间也很重要，相比于两个月后推出完美的产品，有时尽快推出一个不尽完美的产品更好。

在 Web 开发中，质量可以分解为四个维度：

- *到达率*

  到达率是指产品的市场普及程度，即查看网站或使用服务的人数。到达率和盈利能力是正相关关系：访问网站的人越多，购买产品或服务的人就越多。从开发的角度来看，搜索引擎优化（SEO）对到达率的影响最大，所以我们会在 QA 方案里包含 SEO。

- *功能*

  人们一旦访问了你的网站或使用了你的服务，能否把用户留下很大程度上取决于网站功能的质量：一个能像广告宣传那样工作的网站更有可能吸引回头客。与其他几个维度不同，功能测试一般都可以自动执行。

- *可用性*

  功能关心的是功能的正确性，而可用性评估的是人机交互（HCI）。根本问题是：“这个功能是以对目标受众有用的方式交付的吗？”这个问题经常被换成“它易用吗？”，尽管追求易用性经常跟灵活性或能力是相对的：程序员眼中的“容易”可能跟不懂技术的用户眼中的“容易”不一样。换句话说，评估可用性时你必须考虑目标受众。因为可用性评估的根本输入是用户，所以可用性评估一般无法自动完成。然而，你的 QA 方案中应该包含用户测试。

- *审美*

  审美是四个维度中最主观的，因此也是跟开发最不相关的一个维度。尽管跟网站审美相关的开发问题没有几个，但 QA 方案中还是应该包括网站审美的常规评审。把网站展示给有代表性的样本受众，看他们是否觉得已经过时，或者是不是没能激起你所期望的

响应。记住，审美具有时间敏感性（审美标准会随着时间而发生变化），并且因人而异（受到某一受众喜爱的东西可能完全激不起其他受众的兴趣）。

尽管这四个维度在 QA 方案中都要涉及，但因为功能测试和 SEO 可以在开发过程中自动完成，所以我们会将这两个维度作为本章的重点内容。

5.2　逻辑与展示

从广义上来讲，网站上有两个“领域”：*逻辑*（经常被叫作“业务逻辑”，因为商业味儿比较浓，所以在这里没用这个词）和*表示*。你可以认为网站的逻辑存在于纯粹的认知领域。比如，在草地鹨旅行社这个案例中，可能会有个规则要求客户必须持有有效驾照才能租用代步车。这是一条基于数据的简单规则：对于每个代步车预定而言，用户需要有一个有效的驾照。这和表示是分开的。或许它只是最后形成的订单页面上的一个检查框，也有可能客户必须提供一个有效驾照编号，然后由草地鹨旅行社确认其是否有效。这个区分很重要，因为逻辑域中的事情应该尽可能简单清晰，而表示域复杂还是简单则视需要而定。表示域还是可用性和审美问题要关注的课题，而业务域则不是。

你应该尽可能地在逻辑和表示之间划出清晰的界限。这有很多种方式，本书将把重点放在 JavaScript 模块对逻辑的封装上。另一方面，表示，将是对 HTML、CSS、多媒体、JavaScript 和 jQuery 之类的前端库的一种结合。

5.3　测试的类型

本书要讨论的测试主要归为两大类：单元测试和集成测试（我认为“系统测试”属于集成测试）。单元测试的粒度非常细，是对单个组件进行测试以确保其功能正确，而集成测试是对多个组件甚至整个系统之间的交互进行测试。

一般而言，单元测试在测试逻辑时更实用，也更恰当（尽管我们在表示域的代码中也会看到很多使用单元测试的实例）。集成测试则在两个领域中都有用。

5.4　QA技术概览

本书会用以下这些技术和软件进行全面的测试：

- *页面测试*

  页面测试，顾名思义，用来测试页面的表示和前端功能。这同时涉及单元测试和集成测试。我们会用 Mocha 进行页面测试。

- 跨页测试

  跨页测试是对从一个页面转到另一个页面的功能的测试。比如电子商务网站上的结账功能，通常要跨越多个页面。因为这种测试会涉及多个组件，所以一般被当作集成测试。这个测试用的是 Zombie.js。

- 逻辑测试

  逻辑测试会对逻辑域进行单元和集成测试。它只会测试 JavaScript，跟所有表示功能分开。

- 去毛

  去毛不是要找错误，而是要找潜在的错误。去毛的一般概念是找出可能有错误的区域，或者可能在将来导致错误发生的问题代码。我们会用 JSHint 做去毛。

- 链接检查

  链接检查（确保你的网站上没有破损的链接）属于“唾手可得”的那一类测试。对简单的项目做链接检查看起来可能没有必要，但简单项目也会发展成复杂项目，破损的链接也将会出现。越早把链接检查放到 QA 过程里越好。链接检查属于单元测试（链接有效或者无效）。我们会用 LinkChecker 做链接检查。

5.5 运行你的服务器

本章中的所有技术都假定你的网站是处在运行中的。直到目前为止，我们都是用命令 `node meadowlark.js` 手工运行网站。这项技术很简单，我一般会在桌面上专门开一个窗口来做这个工作。然而这并不是唯一的选择。如果你发现自己在修改 JavaScript 时会忘记重启服务器，或许你希望找一个监控工具，在它发现 JavaScript 被修改后会自动重启服务器。nodemon（https://npmjs.org/package/nodemon）非常受欢迎，并且它还有一个 Grunt 插件（https://www.npmjs.org/package/grunt-nodemon）。本章最后还会介绍更多有关 Grunt 的知识。现在，我只是建议你在一个不同的窗口中一直运行你的应用程序。

5.6 页面测试

对于页面测试，我建议把测试真正嵌入到页面中。这样做的优点是在做一个页面时，在浏览器中一加载页面就可以马上发现所有错误。这需要做些设置，我们开始吧。

首先我们需要一个测试框架，这里用的是 Mocha。我们先把这个包添加到项目中：

```
npm install --save-dev mocha
```

注意，我们用的是 `--save-dev` 而不是 `--save`，这是告诉 npm 要把这个包放在开发依赖项中，不要放在运行时依赖项里。这样当我们部署网站的现场实例时，可以减少项目的依赖项。

因为 Mocha 要在浏览器中运行，所以我们要把 Mocha 资源放在 public 目录下，以便让客户端访问到。我们会把这些资源放在子目录 public/vendor 中：

```
mkdir public/vendor
cp node_modules/mocha/mocha.js public/vendor
cp node_modules/mocha/mocha.css public/vendor
```

> 把你用到的第三方库放在一个特殊的目录中是个好主意，比如 vendor。这样比较容易分清哪些代码是需要你负责测试和修改的，哪些代码你不应该触碰。

测试通常需要一个 assert（或 expect）函数。Node 框架中有这个函数，但浏览器中没有，所以我们要用 Chai 断言库：

```
npm install --save-dev chai
cp node_modules/chai/chai.js public/vendor
```

现在有了必需的文件，我们可以修改草地鹨旅行社网站来运行测试了。问题是我们不希望测试一直运行：它不仅会拖慢网站的速度，而且用户也不想看到测试结果。默认情况下测试应该是禁用的，但应该非常容易启用。为了满足这两个目标，我们准备用一个 URL 参数来打开测试。等我们做好之后，访问 http://localhost:3000 会加载首页，而 http://localhost:3000?test=1 将会加载包含测试的首页。

我们准备用一些中间件来检测查询字符串中的 test=1。它必须出现在我们定义的所有路由之前：

```
app.use(function(req, res, next){
        res.locals.showTests = app.get('env') !== 'production' &&
                req.query.test === '1';
        next();
});

// 路由放在这里
```

在后面的章节中，你会更加清晰地了解到这段代码的作用。现在你只需要知道，如果 test=1 出现在任何页面的查询字符串中（并且不是运行在生产服务器上），属性 res.locals.showTests 就会被设为 true。res.locals 对象是要传给视图的上下文的一部分（第 7 章会详细解释）。

现在我们可以修改 views/layouts/main.handlebars，有条件地引入测试框架。修改 <head> 部分：

```
<head>
        <title>Meadowlark Travel</title>
        {{#if showTests}}
                <link rel="stylesheet" href="/vendor/mocha.css">
        {{/if}}
        <script src="//code.jquery.com/jquery-2.0.2.min.js"></script>
</head>
```

这里还用到了 jQuery，因为我们不仅可以用它做网站的主 DOM 处理库，还可以做测试断言。你可以用自己喜欢的任何库（或者根本不用），但我建议你用 jQuery。你应该经常听说 JavaScript 库应该最后加载，放在结束标签 </body> 之前。这种说法是有道理的，我们也会学一些技术使之成为可能，但现在我们要早点儿引入 jQuery[1]。

然后在紧挨着结束标签 </body> 之前：

```
        {{#if showTests}}
                <div id="mocha"></div>
                <script src="/vendor/mocha.js"></script>
                <script src="/vendor/chai.js"></script>
                <script>
                        mocha.ui('tdd');
                        var assert = chai.assert;
                </script>
                <script src="/qa/tests-global.js"></script>
                {{#if pageTestScript}}
                        <script src="{{pageTestScript}}"></script>
                {{/if}}
                <script>mocha.run();</script>
        {{/if}}
</body>
```

注意，我们引入了 Mocha 和 Chai，还有一个 /qa/global-tests.js 脚本。就如它的名字里暗示的那样，这是每个页面上都要运行的测试。在后续继续深入时，我们会有选择地链接每个页面特有的测试，这样你就可以针对不同的页面做不同的测试。我们先从全局测试开始，然后再增加针对各个页面的测试。我们先从单一的、简单的测试开始，确保页面具有有效的标题。创建目录 public/qa，然后在其中创建文件 tests-global.js：

```
suite('Global Tests', function(){
        test('page has a valid title', function(){
              assert(document.title && document.title.match(/\S/) &&
                      document.title.toUpperCase() !== 'TODO');
         });
});
```

注 1：记住性能调优的第一条原则：先测量，再调优。

Mocha 支持多种“界面”来控制测试的风格。默认界面是行为驱动开发（BDD），它让你以行为的方式思考。在 BDD 中，你描述组件和它们的行为，然后用测试去验证这些行为。然而，我发现这些测试经常不适合这一模型，然后 BDD 语言看起来就显得很奇怪。测试驱动开发（TDD）更具可行性，你描述的是测试集和其中的测试。你可以使用两种界面进行自己的测试，但会造成配置上的困难。因此我在本书中坚持使用 TDD。如果你喜欢 BDD，或者 BDD 和 TDD 混合的风格，当然也可以。

接下来运行网站。访问首页并检查下源码，你看不到任何测试相关的代码。把 `test=1` 添加到查询字符串后面（http://localhost:3000/?test=1），你将看到在页面上运行的测试。无论什么时候，当你想测试网站时，只要在查询字符串上加上 `test=1` 就行了。

接下来我们添加针对页面的测试。比如我们想确保关于页面上总是有一个指向联系我们页面的链接。创建一个 public/qa/tests-about.js 文件：

```
suite('"About" Page Tests', function(){
        test('page should contain link to contact page', function(){
                assert($('a[href="/contact"]').length);
         });
});
```

我们还要做最后一件事：在路由中指明视图应该使用哪个页面测试文件。在 meadowlark.js 中修改关于页面的路由：

```
app.get('/about', function(req, res) {
         res.render('about', {
                 fortune: fortune.getFortune(),
                 pageTestScript: '/qa/tests-about.js'
         });
});
```

加载带查询字符串 `test=1` 的关于页面，你将会看到两个测试集并伴随着一次失败。现在添加一个指向尚不存在的联系我们页面的链接，你刷新页面后就能看到测试成功了。

根据网站的属性，你或许想让这个测试更加自动化。比如说，如果你的路由是 /foo，可以自动将针对页面的测试设为 /foo/tests-foo.js。这种方式的不足是不够灵活。比如说，如果你有多个路由指向相同的视图，甚至是非常相似的内容，你可能想要使用同一个测试文件。

现在先克制一下自己想要添加更多测试的欲望，伴随着本书的进程它会不断被添加。现在我们已经有了添加全局和针对页面的测试所必需的框架。

5.7 跨页测试

跨页测试更有挑战性，因为需要你控制和观测浏览器。我们来看一个跨页测试情境的例子。比如，你的网站上有一个包含联系表单的 Request Group Rate 页面。营销部门想知道客户是从哪个页面点击链接进入 Request Group Rate 页面的，他们想知道客户是否在查看胡德河之旅或者俄勒冈海岸退潮。关联上它需要有一些隐藏的表单域和 JavaScript，并且测试将会涉及进入一个页面，然后点击 Request Group Rate 并验证隐藏域是否正确填充了。

我们把这个情境设置好，然后看看如何进行测试。首先我们要创建一个旅游线路的页面，views/tours/hood-river.handlebars：

```
<h1>Hood River Tour</h1>
<a class="requestGroupRate"
        href="/tours/request-group-rate">Request Group Rate.</a>
```

以及一个引用页面，views/tours/request-group-rate.handlebars：

```
<h1>Request Group Rate</h1>
<form>
        <input type="hidden" name="referrer">
        Name: <input type="text" id="fieldName" name="name"><br>
        Group size: <input type="text" name="groupSize"><br>
        Email: <input type="email" name="email"><br>
        <input type="submit" value="Submit">
</form>
<script>
        $(document).ready(function(){
                $('input[name="referrer"]').val(document.referrer);
        });
</script>
```

然后在 meadowlark.js 中为这些页面创建路由：

```
app.get('/tours/hood-river', function(req, res){
         res.render('tours/hood-river');
});
app.get('/tours/request-group-rate', function(req, res){
         res.render('tours/request-group-rate');
});
```

现在我们有了可以测试的对象，还需要测试它的方法，事情从这里开始变得复杂了。要测试这个功能，我们真的需要一个浏览器，或者非常类似浏览器的东西。很显然，我们可以手动在浏览器中访问 /tours/hood-river 页面，然后点击 Request Group Rate 链接，再探查隐藏的表单元素，看看它是否正确填上了引用页，但这么做太麻烦了，我们希望它可以自动完成。

我们要找的是一个被称为*无头浏览器*的东西。无头浏览器意味着这个浏览器不需要真的在

屏幕上显示什么，但它必须表现得像个浏览器。目前有三种流行的解决方案：Selenium、PhantomJS 和 Zombie。Selenium 超级健壮，有丰富的测试支持，但配置它超出了本书的范围。PhantomJS 是一个伟大的项目，并且它确实提供了一个无头 Webkit 浏览器（跟 Chrome 和 Safari 用的是相同的引擎），所以跟 Selenium 一样，它也呈现出了非常高水平的现实性。然而它还没提供我们所需的简单的测试断言，这样我们就只剩下 Zombie 了。

Zombie 没有使用已有的浏览器引擎，所以它不适合用来测试浏览器的功能特性，但用它来测试基本功能是非常好的，这正是我们所需要的。可惜 Zombie 现在不支持 Windows（可以装在 Cygwin 环境下）。然而人们已经在使用它了，在 Zombie 首页（http://zombie.labnotes.org/）上有相关信息。我努力想让本书中的内容与平台无关，但目前还没有 Windows 下的无头浏览器测试方案。如果你是在 Windows 下开发，我建议你看看 Selenium 或 PhantomJS，尽管学起来有一定的难度，但这些项目提供了很多东西。

首先，我们安装 Zombie：

```
npm install --save-dev zombie
```

接下来创建一个新目录，简单地称其为 qa（跟 public/qa 区分开）。在这个目录下创建 qa/tests-crosspage.js 文件：

```
var Browser = require('zombie'),
        assert = require('chai').assert;

var browser;

suite('Cross-Page Tests', function(){

        setup(function(){
                browser = new Browser();
        });

        test('requesting a group rate quote  from the hood river tour page' +
                'should populate the referrer field', function(done){
                var referrer = 'http://localhost:3000/tours/hood-river';
                browser.visit(referrer, function(){
                        browser.clickLink('.requestGroupRate', function(){
                                assert(browser.field('referrer').value
                                        === referrer);
                                done();
                        });
                });
        });

        test('requesting a group rate from the oregon coast tour page should ' +
                        'populate the referrer field', function(done){
                var referrer = 'http://localhost:3000/tours/oregon-coast';
                browser.visit(referrer, function(){
                        browser.clickLink('.requestGroupRate', function(){
                                assert(browser.field('referrer').value
```

```
                                        === referrer);
                                done();
                        });
                });
        });

        test('visiting the "request group rate" page dirctly should result ' +
                        'in an empty referrer field', function(done){
                browser.visit('http://localhost:3000/tours/request-group-rate',
                        function(){
                                assert(browser.field('referrer').value === '');
                                done();
                });
        });
});
```

setup 的参数是一个函数，测试框架运行每个测试之前都会执行它，我们在这里为每个测试创建一个新的浏览器实例。我们有三个测试。前两个检查如果你来自产品页面，引用页是否正确。方法 browser.visit 会真正加载页面，页面加载完成后，就会调用回调函数。然后用方法 browser.clickLink 找到 class 为 requestGroupRate 的链接，并访问它。链接目标页面加载完后调用回调函数，我们就到了 Request Group Rate 页面上。剩下唯一要做的就是断言隐藏域 referrer 跟我们原来访问的页面是匹配的。browser.field 方法会返回一个 DOM 元素对象，具有 value 属性。最后一个测试只是确保直接访问 Request Group Rate 页面时 referrer 为空。

在进行测试之前，必须先启动服务器（node meadowlark.js）。你应该在另一个窗口中启动它，以便看到控制台错误。然后，运行测试看看我们做得怎么样（确保你有全局安装的 Mocha：npm install -g mocha）：

```
mocha -u tdd -R spec qa/tests-crosspage.js 2>/dev/null
```

我们将看到有一个测试失败了。失败的是俄亥俄海滩之旅的页面，这一点也不意外，因为我们还没有做那个页面。但另外两个测试通过了，所以我们的测试是可以用的。继续添加俄亥俄海滩之旅的页面，所有测试就都能通过了。注意前面那个命令，我们用的是 TDD 界面（默认是 BDD），还用了一个叫 spec 的报告。spec 报告比默认报告提供的信息要多一些。（等你有上百个测试的时候，你可能还是想用默认报告。）最后，你可能会注意到我们扔掉了错误输出（2>/dev/null）。Mocha 会报告失败测试的全部堆栈跟踪。这些信息可能有用，但一般你只想看到哪些测试通过了，哪些失败了。如果你需要更多信息，去掉 2>/dev/null 就能看到错误的细节了。

在实现功能特性之前写测试有一个优点（如果测试正确的话），即它们一开始都会失败。当你看着自己的测试开始通过时，不仅能得到满足感，还能确保测试是正确的。如果在你还没实现任何功能特性时测试就能通过，那这个测试很可能是有问题的。有时这被称为“红灯，绿灯”测试。

5.8 逻辑测试

我们还要用 Mocha 做逻辑测试。现在我们只有一个小小的功能（幸运饼干生成器），所以设置它相当容易。另外，因为我们只有一个组件，也不能做集成测试，所以我们只添加单元测试。创建文件 qa/tests-unit.js：

```
var fortune = require('../lib/fortune.js');
var expect = require('chai').expect;

suite('Fortune cookie tests', function(){

    test('getFortune() should return a fortune', function(){
        expect(typeof fortune.getFortune() === 'string');
    });
});
```

现在我们可以运行 Mocha 来进行这个新的测试集：

```
mocha -u tdd -R spec qa/tests-unit.js
```

虽不是特别激动人心，但它为我们提供了一个模板，本书后续测试都可以照此实现。

> 测试熵功能（随机的功能）很有挑战性。我们能对幸运饼干生成器做的另一个测试是确保它返回了一个随机的幸运饼干。但你怎么知道某个东西是否是随机的呢？一种方式是获取数量庞大的幸运饼干，比如 1000 个，然后测量响应的分布情况。如果函数确实是随机的，那就不会有突出的响应。这种方式的缺点是它的不确定性：某个幸运饼干出现的频率有可能（但不太可能）比其他的幸运饼干多 10 倍。如果这种情况出现了，测试可能失败（这要取决于你给随机设定的阈值有多激进），但实际上那或许并不能表明所测试的系统是失败的，它只是测试熵系统的一种结果。具体到我们的幸运饼干生成器，可能生成 50 个饼干，至少有三种不同的就是合理的。另一方面，如果我们是为科学模拟或安全组件开发随机源，可能要做更详细的测试。我们要说的重点是测试熵功能很困难，需要多思考。

5.9 去毛

好的去毛机就像第二双眼睛，它能发现被我们人类大脑忽略的东西。最早的 JavaScript 去毛机是 Douglas Crockford 的 JSLint。Anton Kovalyov 在 2011 年创建了 JSLint 的分支，于是 JSHint 诞生了。Kovalyov 认为 JSLint 过于坚持己见了，所以他想创建一个定制性更强的、由社区制定的 JavaScript 去毛机。尽管我同意 Crockford 的几乎全部去毛建议，但我更喜欢能定制的去毛机，因此我推荐使用 JSHint[1]。

注 1：Nicholas Zakas 的 ESLint（http://eslint.org/）也是不错的选择。

通过 npm 获取 JSHint 非常容易：

```
npm install -g jshint
```

运行它也非常简单，只要指定源文件名调用它就可以了：

```
jshint meadowlark.js
```

如果你是一直跟着我们做的，JSHint 应该不会对 meadowlark.js 有任何抱怨。要看 JSHint 能帮你做什么，请把下面这行代码放到 meadowlark.js 中，然后再像前面那样运行 JSHint：

```
if( app.thing == null ) console.log( 'bleat!' );
```

（JSHint 会抱怨你用了 == 而不是 ===，而 JSLint 还会抱怨缺少大括号。）

我向你保证，坚持用去毛机能让你变成更优秀的程序员。既然如此，如果能把去毛机集成到编辑器中，以便在你刚犯下错误时就能提醒你，这样岂不更好？你是幸运的，因为 JSHint（http://www.jshint.com/install/）能够集成到很多流行的编辑器中。

5.10 链接检查

检查死链接看起来没什么吸引力，但它对搜索引擎如何给你的网站评级却有巨大的影响。它很容易集成到你的工作流中，所以不这样做就太不明智了。

我推荐用 LinkChecker（http://wummel.github.io/linkchecker/）。它是跨平台的，既有命令行界面，也有图形界面。只要装上它并指向你的首页就可以了：

```
linkchecker http://localhost:3000
```

我们的网站还没有太多页面，所以 LinkChecker 应该很快就能检查完。

5.11 用Grunt实现自动化

我们在用的 QA 工具，如测试套件、去毛和链接检查器，只有在真正使用时才有价值。很多 QA 方案就是因为未使用而枯萎直至死去。如果你必须记住 QA 工具链中的所有组件和所有运行它们的命令，你（或你共事的其他开发人员）很有可能渐渐不再使用它们了。如果你准备花时间去掌握一个完备的 QA 工具链，那是不是也值得花点儿时间把这个过程自动化，把这个工具链真正用起来呢？

我们很幸运，一个叫 Grunt 的工具可以很容易地实现这些任务的自动化。我们将把逻辑测试、跨页测试、去毛和链接检查放到一个 Grunt 命令中。为什么没有页面测试呢？尽管用 PhantomJS 或 Zombie 之类的无头浏览器也有可能做到，但配置复杂，并且也超出了本书的

范围。更进一步说，浏览器测试通常被设计成好像你运行在单个页面上，所以把它们合到其他测试中也没太大价值。

首先要装上 Grunt 命令行以及 Grunt 本身：

```
sudo npm install -g grunt-cli
npm install --save-dev grunt
```

Grunt 要靠插件完成任务，Grunt 插件列表（http://gruntjs.com/plugins）中列出了所有可用插件。我们需要 Mocha、JSHint 和 LinkChecker 的插件。在写本书时，还没有 LinkChecker 的插件，所以我们只能用执行 shell 命令的通用插件。接下来我们先把必需的插件装上：

```
npm install --save-dev grunt-cafe-mocha
npm install --save-dev grunt-contrib-jshint
npm install --save-dev grunt-exec
```

现在所有插件都装好了，在项目目录下创建一个 Gruntfile.js 文件：

```
module.exports = function(grunt) {

        // 加载插件
        [
                'grunt-cafe-mocha',
                'grunt-contrib-jshint',
                'grunt-exec',
        ].forEach(function(task){
                grunt.loadNpmTasks(task);
        });

        // 配置插件
        grunt.initConfig({
                cafemocha: {
                        all: { src: 'qa/tests-*.js', options: { ui: 'tdd' }, }
                },
                jshint: {
                        app: ['meadowlark.js', 'public/js/**/*.js',
                                'lib/**/*.js'],
                        qa: ['Gruntfile.js', 'public/qa/**/*.js', 'qa/**/*.js'],
                },
                exec: {
                        linkchecker:
                                { cmd: 'linkchecker http://localhost:3000' }
                },
        });
        // 注册任务
        grunt.registerTask('default', ['cafemocha','jshint','exec']);
};
```

在“加载插件”部分，我们指定了要用哪些插件，跟我们通过 npm 安装的插件一样。因为我不喜欢一次次地重复输入 `loadNpmTasks`（一旦你开始依赖 Grunt，相信我，你会添加更多插件的），所以我选择把它们全部放到数组中，并用 `forEach` 循环遍历。

在“配置插件”部分，我们必须做些工作让每个插件都能正常工作。对于 `cafemocha` 插件（由它运行逻辑和跨页测试），我们必须告诉它测试在哪里。我们把所有测试都放在子目录 qa 下面，并在文件名中加上前缀 tests-。注意，我们必须指定 TDD 界面。如果是 TDD 和 BDD 混合的界面，则必须想办法把它们分开。比如，你可以用两个前缀 tests-tdd- 和 tests-bdd-。

对于 JSHint，我们必须指定要对哪些 JavaScript 文件去毛。这里一定要当心！依赖项经常不一定能通过 JSHint，或者它们用的是不同的 JSHint 设置，并且你会被 JSHint 错误淹没，而其中很多代码都不是你写的。具体来说，你要确保别把 node_modules 目录以及任何 vendors 目录包含在内。目前 `grunt-contrib-jshint` 还不能排除文件，只能包含它们。所以我们必须指定所有想要包含在内的文件。我一般会把想要包含的文件分成两个列表：真正构成应用程序或网站的 JavaScript，以及 QA JavaScript。它们都要去毛，但这样分开更容易管理一些。注意通配符 `/**/` 的含义是“子目录中的所有文件”。尽管现在还没有 public/js 目录，但我们会有的。隐含着排除的是 node_modules 和 public/vendor 目录。

最后，我们配置了 `grunt-exec` 插件，让它运行 LinkChecker。注意，我们把端口 3000 硬编码在这个插件的配置里了。这最好能参数化，我把这当作练习留给读者了[1]。

最后我们“注册”了这些任务：把单个的插件放到一个命名分组中。一个特定名称的任务 `default`，在你只是输入 `grunt` 后，就会默认运行。

现在你只需确保服务器在（后台或另一个窗口中）运行着，然后运行 Grunt：

```
grunt
```

所有测试都会运行（除了页面测试），所有代码都会去毛，所有链接都会被检查！如果某个组件失效，Grunt 会给出错误消息并终止，否则它会报告“完成，没有错误”。没有什么比看到这条消息更让人满意的了，所以养成提交前运行 Grunt 的习惯吧！

5.12 持续集成

我要向你介绍一个极其实用的 QA 概念：持续集成（CI）。如果你在团队中工作，它尤其重要，但即便你只是一个人在战斗，它也能为你提供一些不可或缺的纪律。基本上你每次向共享服务器贡献代码时，CI 都会运行部分或全部测试。如果所有测试都通过了，通常什么也不会发生（你可能会收到一封邮件说“干得好”，这取决于你是如何配置 CI 的）。另一方面，如果有测试失败了，后果一般是更加公开。这也是取决于你是如何配置 CI 的，但一般整个团队都会收到一封邮件说你“搞砸了构建”。如果你们的集成管理员是个虐待狂，有时老板也会出现在邮件列表中。我听说甚至有的团队会在有人搞砸构建时设置灯光

注 1：入手请参见 grunt.option 文档（http://gruntjs.com/api/grunt.option）。

和警报器，并且在一个特别有创造性的办公室，一个微型的机器人泡沫导弹发射装置会向犯错的开发人员发射泡沫塑料弹。它是一个提交前运行 QA 工具链的强力激励措施。

CI 服务器的安装和配置超出了本书的范围，但如果不介绍它，这一章就不能算是完整的 QA 章节。目前 Node 中最流行的 CI 服务器是 Travis CI（http://about.travis-ci.org/docs/user/getting-started）。Travis CI 是一个托管的解决方案，非常有吸引力（省去了自己设置 CI 服务器的麻烦）。如果你用 GitHub，它提供了卓越的集成支持。非常成熟的 CI 服务器 Jenkins 现在也有 Node 插件（https://wiki.jenkins-ci.org/display/JENKINS/NodeJS+Plugin）。JetBrains 卓越的 TeamCity（http://www.jetbrains.com/teamcity/）现在也提供 Node 插件。

如果你是独立做项目，CI 服务器对你的帮助可能不是特别大，但如果你在团队中工作，或在做一个开源项目，我强烈推荐给项目设置 CI。

第6章

请求和响应对象

在用 Express 构建 Web 服务器时，大部分工作都是从请求对象开始，到响应对象终止。这两个对象起源于 Node，Express 对其进行了扩展。在深入探讨这两个对象之前，我们先了解一点背景知识，看看客户端（通常是浏览器）是如何向服务器请求一个页面，以及页面是如何返回的。

6.1　URL的组成部分



- 协议

  协议确定如何传输请求。我们主要是处理 http 和 https。其他常见的协议还有 file 和 ftp。

- 主机名

  主机名标识服务器。运行在本地计算机（localhost）和本地网络的服务器可以简单地表示，比如用一个单词，或一个数字 IP 地址。在 Internet 环境下，主机名通常以一个顶级域名（TLD）结尾，比如 .com 或 .net。另外，也许还会有子域名作为主机名的前缀。

子域名可以是任何形式的，其中 www 最为常见。子域名通常是可选的。

- *端口*

  每一台服务器都有一系列端口号。一些端口号比较“特殊”，如 80 和 443 端口。如果省略端口值，那么默认 80 端口负责 HTTP 传输，443 端口负责 HTTPS 传输。如果不使用 80 和 443 端口，就需要一个大于 1023[1] 的端口号。通常使用容易记忆的端口号，如 3000、8080 或 8088。

- *路径*

  URL 中影响应用程序的第一个组成部分通常是路径（在考虑协议、主机名和端口的基础上做决定很合理，但是不够好）。路径是应用中的页面或其他资源的唯一标识。

- *查询字符串*

  查询字符串是一种键值对集合，是可选的。它以问号（?）开头，键值对则以与号（&）分隔开。所有的名称和值都必须是 URL 编码的。对此，JavaScript 提供了一个嵌入式的函数 `encodeURIComponent` 来处理。例如，空格被加号（+）替换。其他特殊字符被数字型字符替换。

- *信息片段*

  信息片段（或*散列*）被严格限制在浏览器中使用，不会传递到服务器。用它控制单页应用或 AJAX 富应用越来越普遍。最初，信息片段只是用来让浏览器展现文档中通过锚点标记（`<a id="chapter06">`）指定的部分。

6.2　HTTP请求方法

HTTP 协议确定了客户端与服务器通信的*请求方法*集合（通常称为 HTTP verbs）。很显然，`GET` 和 `POST` 最为常见。

在浏览器中键入一个 URL（或点击一个链接），服务器会接收到一个 HTTP `GET` 请求，其中的重要信息是 URL 路径和查询字符串。至于如何响应，则需要应用程序结合方法、路径和查询字符串来决定。

对于一个网站来说，大部分页面都响应 `GET` 请求。`POST` 请求通常用来提交信息到服务器后台（例如表单处理）。服务器将请求中包含的所有信息（例如表单）处理完成之后，用以响应的 HTML 通常与相应的 `GET` 请求是一样的。与服务器通信时，浏览器只使用 `GET` 和 `POST` 方法（如果没有使用 AJAX）。

另一方面，网络服务通常会使用更多的创造性 HTTP 方法。例如，一个 HTTP 方法被命名

注 1：0~1023 端口为“知名端口”。

为 DELETE，它就用来接受 API 指令执行删除功能。

使用 Node 和 Express，可以完全掌控响应方法（尽管一些更复杂的方法支持得不是很好）。在 Express 中，通常要针对特殊方法编写处理程序。

6.3 请求报头

我们浏览网页时，发送到服务器的并不只是 URL。当你访问一个网站时，浏览器会发送很多“隐形”信息。这里讨论的并不是个人信息泄露问题（浏览器被恶意软件侵染时会出现这个问题）。服务器会因此得知优先响应哪种语言的页面（例如，在西班牙下载 Chrome 浏览器，如果有西班牙语的版本，就会接收到一个西班牙语的访问页面）。它也会发送“用户代理”信息（浏览器、操作系统和硬件设备）和其他一些信息。所有能够确保你了解请求对象头文件属性的信息都将会作为请求报头发送。如果想查看浏览器发送的信息，可以创建一个非常简单的 Express 路由来展示一下：

```
app.get('/headers', function(req,res){
    res.set('Content-Type','text/plain');
    var s = '';
    for(var name in req.headers) s += name + ': ' + req.headers[name] + '\n';
    res.send(s);
});
```

6.4 响应报头

正如浏览器以请求报头的形式发送隐藏信息到服务器，当服务器响应时，同样会回传一些浏览器没必要渲染和显示的信息，通常是元数据和服务器信息。我们已经熟悉内容类型头信息，它告诉浏览器正在被传输的内容类型（网页、图片、样式表、客户端脚本等）。特别要注意的是，不管 URL 路径是什么，浏览器都根据内容类型报头处理信息。因此你可以通过一个叫作 /image.jpg 的路径提供网页，也可以通过一个叫作 /text.html 的路径提供图片。（这样做并不合情理，这里要讲的重点是路径是抽象的，浏览器只根据内容类型来决定内容该如何渲染。）除了内容类型之外，报头还会指出响应信息是否被压缩，以及使用的是哪种编码。响应报头还可以包含关于浏览器对资源缓存时长的提示。优化网站时需要着重考虑这一点，我们将在第 16 章详细讨论。响应报头还经常会包含一些关于服务器的信息，一般会指出服务器的类型，有时甚至会包含操作系统的详细信息。返回服务器信息存在一个问题，那就是它会给黑客一个可乘之机，从而使站点陷入危险。非常重视安全的服务器经常忽略此信息，甚至提供虚假信息。禁用 Express 的 X-Powered-By 头信息很简单：

```
app.disable('x-powered-by');
```

在浏览器的开发者工具中可以找到响应报头信息。例如，在 Chrome 浏览器中查看响应报头信息的操作如下：

(1) 打开控制台。
(2) 点击网络标签页。
(3) 重新载入页面。
(4) 在请求列表中选取网页（通常是第一个）。
(5) 点击报头标签页，你就可以看到所有响应报头信息了。

6.5 互联网媒体类型

内容类型报头信息极其重要，没有它，客户端很难判断如何渲染接收到的内容。内容类型报头就是一种*互联网媒体类型*，由一个类型、一个子类型以及可选的参数组成。例如，`text/html;charset=UTF-8` 说明类型是 text，子类型是 html，字符编码是 UTF-8。互联网编号分配机构维护了一个官方的互联网媒体类型清单（http://www.iana.org/assignments/media-types/media-types.xhtml）。我们常见的 content type、Internet media type 和 MIME type 是可以互换的。MIME（多用途互联网邮件扩展）是互联网媒体类型的前身，它们大部分是相同的。

6.6 请求体

除请求报头外，请求还有一个*主体*（就像作为实际内容返回的响应主体一样）。一般 `GET` 请求没有主体内容，但 `POST` 请求是有的。`POST` 请求体最常见的媒体类型是 `application/x-www-form-urlendcoded`，是键值对集合的简单编码，用 & 分隔（基本上和查询字符串的格式一样）。如果 `POST` 请求需要支持文件上传，则媒体类型是 `multipart/form-data`，它是一种更为复杂的格式。最后是 AJAX 请求，它可以使用 `application/json`。

6.7 参数

“参数”这个词可以有很多种解释，它通常是困惑的源头。对于任何一个请求，参数可以来自查询字符串、会话（请求 cookies，详见第 9 章）、请求体或指定的路由参数（详见第 14 章）。在 Node 应用中，请求对象的参数方法会重写所有的参数。因此我们最好不要深究。通常这会带来问题，一个参数在查询字符串中，另一个在 `POST` 请求体中或会话中，哪个会赢呢？这会产生让人抓狂的 bug。PHP 是产生这种混乱的主要原因：为了尽量“方便”，它将所有参数重新写入了一个称为 `$_REQUEST` 的变量，由于某种原因，人们曾认为这是个前所未有的好主意。我们将学习保存不同类型参数的专用属性，我认为这能够减少困惑。

6.8 请求对象

请求对象（通常传递到回调方法，这意味着你可以随意命名，通常命名为 `req` 或 `request`）

的生命周期始于 Node 的一个核心对象 `http.IncomingMessage` 的实例。Express 添加了一些附加功能。我们来看看请求对象中最有用的属性和方法（除了来自 Node 的 `req.headers` 和 `req.url`，所有这些方法都由 Express 添加）。

- `req.params`

  一个数组，包含命名过的路由参数。我们将在第 14 章进行详细介绍。

- `req.param(name)`

  返回命名的路由参数，或者 `GET` 请求或 `POST` 请求参数。建议你忽略此方法。

- `req.query`

  一个对象，包含以键值对存放的查询字符串参数（通常称为 `GET` 请求参数）。

- `req.body`

  一个对象，包含 `POST` 请求参数。这样命名是因为 `POST` 请求参数在 `REQUEST` 正文中传递，而不像查询字符串在 URL 中传递。要使 `req.body` 可用，需要中间件能够解析请求正文内容类型，我们将在第 10 章进行详细介绍。

- `req.route`

  关于当前匹配路由的信息。主要用于路由调试。

- `req.cookies/req.singnedCookies`

  一个对象，包含从客户端传递过来的 cookies 值。详见第 9 章。

- `req.headers`

  从客户端接收到的请求报头。

- `req.accepts([types])`

  一个简便的方法，用来确定客户端是否接受一个或一组指定的类型（可选类型可以是单个的 MIME 类型，如 `application/json`、一个逗号分隔集合或是一个数组）。写公共 API 的人对该方法很感兴趣。假定浏览器默认始终接受 HTML。

- `req.ip`

  客户端的 IP 地址。

- `req.path`

  请求路径（不包含协议、主机、端口或查询字符串）。

- `req.host`

  一个简便的方法，用来返回客户端所报告的主机名。这些信息可以伪造，所以不应该用于安全目的。

- `req.xhr`

  一个简便属性，如果请求由 Ajax 发起将会返回 `true`。

- `req.protocol`

  用于标识请求的协议（`http` 或 `https`）。

- `req.secure`

  一个简便属性，如果连接是安全的，将返回 `true`。等同于 `req.protocol==='https'`。

- `req.url/req.originalUrl`

  有点用词不当，这些属性返回了路径和查询字符串（它们不包含协议、主机或端口）。`req.url` 若是出于内部路由目的，则可以重写，但是 `req.orginalUrl` 旨在保留原始请求和查询字符串。

- `req.acceptedLanguages`

  一个简便方法，用来返回客户端首选的一组（人类的）语言。这些信息是从请求报头中解析而来的。

6.9 响应对象

响应对象（通常传递到回调方法，这意味着你可以随意命名它，通常命名为 `res`、`resp` 或 `response`）的生命周期始于 Node 核心对象 `http.ServerResponse` 的实例。Express 添加了一些附加功能。我们来看看响应对象中最有用的属性和方法（所有这些方法都是由 Express 添加的）。

- `res.status(code)`

  设置 HTTP 状态代码。Express 默认为 200（成功），所以你可以使用这个方法返回状态 404（页面不存在）或 500（服务器内部错误），或任何一个其他的状态码。对于重定向（状态码 301、302、303 和 307），有一个更好的方法：`redirect`。

- `res.set(name,value)`

  设置响应头。这通常不需要手动设置。

- `res.cookie（name,vaue,[options]）,res.clearCookie(name,[options])`

  设置或清除客户端 cookies 值。需要中间件支持，详见第 9 章。

- `res.redirect([status],url)`

  重定向浏览器。默认重定向代码是 302（建立）。通常，你应尽量减少重定向，除非永久移动一个页面，这种情况应当使用代码 301（永久移动）。

- `res.send(body),res.send(status,body)`

  向客户端发送响应及可选的状态码。Express 的默认内容类型是 `text/html`。如果你想改为 `text/plain`，需要在 `res.send` 之前调用 `res.set('Content-Type','text/plain\')`。如果 `body` 是一个对象或一个数组，响应将会以 JSON 发送（内容类型需要被正确设置），不过既然你想发送 JSON，我推荐你调用 `res.json`。

- `res.json(json),res.json(status,json)`

  向客户端发送 JSON 以及可选的状态码。

- `res.jsonp(json),req.jsonp(status,json)`

  向客户端发送 JSONP 及可选的状态码。

- `res.type(type)`

  一个简便的方法，用于设置 `Content-Type` 头信息。基本上相当于 `res.set('Content-Type','type')`，只是如果你提供了一个没有斜杠的字符串，它会试图把其当作文件的扩展名映射为一个互联网媒体类型。比如，`res.type('txt')` 会将 `Content-Type` 设为 `text/plain`。此功能在有些领域可能会有用（例如自动提供不同的多媒体文件），但是通常应该避免使用它，以便明确设置正确的互联网媒体类型。

- `res.format(object)`

  这个方法允许你根据接收请求报头发送不同的内容。这是它在 API 中的主要用途，我们将会在第 15 章详细讨论。这里有一个非常简单的例子：`res.format({'text/plain':'hi there','text/html':'<b>hi there</b>'})`。

- `res.attachment([filename]),res.download(path,[filename],[callback])`

  这两种方法会将响应报头 `Content-Disposition` 设为 `attachment`，这样浏览器就会选择下载而不是展现内容。你可以指定 `filename` 给浏览器作为对用户的提示。用 `res.download` 可以指定要下载的文件，而 `res.attachment` 只是设置报头。另外，你还要将内容发送到客户端。

- `res.sendFile(path,[option],[callback])`

  这个方法可根据路径读取指定文件并将内容发送到客户端。使用该方法很方便。使用静态中间件，并将发送到客户端的文件放在公共目录下，这很容易。然而，如果你想根据条件在相同的 URL 下提供不同的资源，这个方法可以派上用场。

- `res.links(links)`

  设置链接响应报头。这是一个专用的报头，在大多数应用程序中几乎没有用处。

- `res.locals,res.render(view,[locals],callback)`

  `res.locals` 是一个对象，包含用于渲染视图的默认上下文。`res.render` 使用配置的模

板引擎渲染视图（不能把 `res.render` 的 `locals` 参数与 `res.locals` 混为一谈，上下文在 `res.locals` 中会被重写，但在没有被重写的情况下仍然可用）。`res.render` 的默认响应代码为 200，使用 `res.status` 可以指定一个不同的代码。视图渲染将在第 7 章深入讨论。

6.10　获取更多信息

由于 JavaScript 的原型继承，有时确切知道自己在做什么是很困难的。Node 提供了 Express 扩展对象，添加的程序包同样也可以扩展它们。有时候弄明白到底什么是可用的是个挑战。通常，我推荐逆向作业：如果你正在寻找某些功能，首先要查看 Express 的 API 文档（http://expressjs.com/api.html）。Express 的 API 相当齐全，你一般都会在这里找到想要的。

如果你需要的信息没在文档中，有时就不得不深入研究 Express 源码（https://github.com/visionmedia/express/tree/master）。我鼓励你这么做，它并没有想象中那么可怕。下面是 Express 源码的路径说明。

- lib/application.js
  Express 主接口。如果想了解中间件是如何接入的，或视图是如何被渲染的，可以看这里。

- lib/express.js
  这是一个相对较短的 shell，是 lib/application.js 中 Connect 的功能性扩展，它返回一个函数，可以用 `http.createServer` 运行 Express 应用。

- lib/request.js
  扩展了 Node 的 `http.IncomingMessage` 对象，提供了一个稳健的请求对象。关于请求对象属性和方法的所有信息都在这个文件里。

- lib/response.js
  扩展了 Node 的 `http.ServerReponse` 对象，提供响应对象。关于响应对象的所有属性和方法都在这个文件里。

- lib/router/route.js
  提供基础路由支持。尽管路由是应用的核心，但这个文件只有不到 200 行，你会发现它非常地简单优雅。

在你深入研究 Express 源码时，或许需要参考 Node 文档（http://nodejs.org/api/http.html），尤其是 `HTTP` 模块部分。

6.11 小结

本章对请求和响应对象作了概述，它们是 Express 应用中不可或缺的组成部分。然而大部分时候我们只需用到其中一小部分。因此，我们要根据使用的频繁程度将其分解开来。

6.11.1 内容渲染

大多数情况下，渲染内容用 `res.render`，它最大程度地根据布局渲染视图。如果想写一个快速测试页，也许会用到 `res.send`。你可以使用 `req.query` 得到查询字符串的值，使用 `req.session` 得到会话值，或使用 `req.cookie/req.singedCookies` 得到 cookies 值。示例 6-1 到示例 6-8 演示了常见的内容渲染任务：

示例 6-1　基本用法

```
// 基本用法
app.get('/about', function(req, res){
        res.render('about');
});
```

示例 6-2　200 以外的响应代码

```
app.get('/error', function(req, res){
        res.status(500);
        res.render('error');
});
// 或是一行……
app.get('/error', function(req, res){
        res.status(500).render('error');
});
```

示例 6-3　将上下文传递给视图，包括查询字符串、cookie 和 session 值

```
app.get('/greeting', function(req, res){
         res.render('about', {
                 message: 'welcome',
                 style: req.query.style,
                 userid: req.cookie.userid,
                 username: req.session.username,
         });
});
```

示例 6-4　没有布局的视图渲染

```
// 下面的 layout 没有布局文件，即 views/no-layout.handlebars
// 必须包含必要的 HTML
app.get('/no-layout', function(req, res){
        res.render('no-layout', { layout: null });
});
```

示例 6-5　使用定制布局渲染视图

```
// 使用布局文件 views/layouts/custom.handlebars
```

```
app.get('/custom-layout', function(req, res){
        res.render('custom-layout', { layout: 'custom' });
});
```

示例 6-6　渲染纯文本输出

```
app.get('/test', function(req, res){
        res.type('text/plain');
        res.send('this is a test');
});
```

示例 6-7　添加错误处理程序

```
// 这应该出现在所有路由方法的结尾
// 需要注意的是，即使你不需要一个 " 下一步 " 方法
// 它也必须包含，以便 Express 将它识别为一个错误处理程序
app.use(function(err, req, res, next){
        console.error(err.stack);
        res.status(500).render('error');
});
```

示例 6-8　添加一个 404 处理程序

```
// 这应该出现在所有路由方法的结尾
app.use(function(req, res){
        res.status(404).render('not-found');
});
```

6.11.2　处理表单

当你处理表单时，表单信息一般在 `req.body` 中（或者偶尔在 `req.query` 中）。你可以使用 `req.xhr` 来判断是 AJAX 请求还是浏览请求（第 8 章将深入讨论）。让我们看看示例 6-9 和示例 6-10。

示例 6-9　基本表单处理

```
// 必须引入中间件 body-parser
app.post('/process-contact', function(req, res){
        console.log('Received contact from ' + req.body.name +
                ' <' + req.body.email + '>');
        // 保存到数据库……
        res.redirect(303, '/thank-you');
});
```

示例 6-10　更强大的表单处理

```
// 必须引入中间件 body-parser
app.post('/process-contact', function(req, res){
        console.log('Received contact from ' + req.body.name +
                ' <' + req.body.email + '>');
        try {
                // 保存到数据库……

                return res.xhr ?
```

```
                res.render({ success: true }) :
                res.redirect(303, '/thank-you');
        } catch(ex) {
                return res.xhr ?
                        res.json({ error: 'Database error.' }) :
                        res.redirect(303, '/database-error');
        }
});
```

6.11.3　提供一个API

如果提供一个类似于表单处理的 API，参数通常会在 `req.query` 中，虽然也可以使用 `req.body`。与其他 API 不同，这种情况下通常会返回 JSON、XML 或纯文本，而不是 HTML。你会经常使用不太常见的 HTTP 方法，比如 `PUT`、`POST` 和 `DELETE`。提供 API 将在第 15 章深入讨论。示例 6-11 和示例 6-12 使用下面的“产品”数组（通常是从数据库中检索）：

```
var tours = [
        { id: 0, name: 'Hood River', price: 99.99 },
        { id: 1, name: 'Oregon Coast', price: 149.95 },
];
```

“节点”一词经常用于描述 API 中的单个方法。

示例 6-11　简单的 GET 节点，只返回 JSON 数据

```
app.get('/api/tours'), function(req, res){
        res.json(tours);
});
```

示例 6-12 根据客户端的首选项，使用 Express 中的 `res.format` 方法对其响应。

示例 6-12　GET 节点，返回 JSON、XML 或 text

```
app.get('/api/tours', function(req, res){
        var toursXml = '<?xml version="1.0"?><tours>' +
                products.map(function(p){
                        return '<tour price="' + p.price +
                                '" id="' + p.id + '">' + p.name + '</tour>';
                }).join('') + '</tours>'';
        var toursText = tours.map(function(p){
                        return p.id + ': ' + p.name + ' (' + p.price + ')';
                }).join('\n');
        res.format({
                'application/json': function(){
                        res.json(tours);
                },
                'application/xml': function(){
```

```
                res.type('application/xml');
                res.send(toursXml);
            },
            'text/xml': function(){
                res.type('text/xml');
                res.send(toursXml);
            }
            'text/plain': function(){
                res.type('text/plain');
                res.send(toursXml);
            }
        });
});
```

在示例 6-13 中，PUT 节点更新一个产品信息然后返回 JSON。参数在查询字符串中传递（路由字符串中的 `'':id''` 命令 Express 在 `req.params` 中增加一个 `id` 属性）。

示例 6-13　用于更新的 PUT 节点

```
//API 用于更新一条数据并且返回 JSON；参数在查询字符串中传递
app.put('/api/tour/:id', function(req, res){
        var p = tours.some(function(p){ return p.id == req.params.id });
        if( p ) {
                if( req.query.name ) p.name = req.query.name;
                if( req.query.price ) p.price = req.query.price;
                res.json({success: true});
        } else {
                res.json({error: 'No such tour exists.'});
        }
});
```

最后，示例 6-14 展示了一个 DEL 节点。

示例 6-14　用于删除的 DEL 节点

```
// API 用于删除一个产品
api.del('/api/tour/:id', function(req, res){
        var i;
        for( var i=tours.length-1; i>=0; i-- )
                if( tours[i].id == req.params.id ) break;
        if( i>=0 ) {
                tours.splice(i, 1);
                res.json({success: true});
        } else {
                res.json({error: 'No such tour exists.'});
        }
});
```

第 7 章

Handlebars模板引擎

如果你没用过模板，或者根本不知道模板是什么，那它就是你将要从这本书中获得的最重要的东西。如果你是一个 PHP 后端开发者，可能会认为 PHP 是第一批模板语言之一这根本没什么大惊小怪的。几乎所有的主流开发语言都为了 Web 开发而增加了模板支持。目前有所不同的是“模板引擎”与开发语言已经解耦。典型的例子是 Mustache，它是一个极受欢迎的、独立于开发语言的模板引擎。

那么到底什么是模板？让我们首先看看模板不是什么，我们以最明显也最直接的方式用一种语言生成另外一种语言（具体来说，我们会使用 JavaScript 生成一些 HTML）：

```
document.write('<h1>Please Don\'t Do This</h1>');
document.write('<p><span class="code">document.write</span> is naughty,\n');
document.write('and should be avoided at all costs.</p>');
document.write('<p>Today\'s date is ' + new Date() + '.</p>');
```

之所以称之为“明显”的唯一的原因也许是，这是一直被教导的编程方式：

```
10 PRINT "Hello world!"
```

在命令式语言中，我们习惯于说“做这个，做那个，然后做另外的”。对于有些事情，这种方法是可行的。如果你有一段 500 行的 JavaScript 代码，执行的是一个复杂计算，然后返回一个数值，并且每一步都是依赖于上一步的，这并不会有什么问题。但如果是另外一种情况呢？假如你有一段 500 行的 HTML 代码和一段 3 行的 JavaScript 代码。写 500 遍 `document.write` 有意义吗？一点也没有。

事实上，问题出现在这里：切换上下文环境是困难的。如果你写了大量的 JavaSctipt，混

合在 HTML 中会引起麻烦和混乱。另一种方法并不糟糕：我们已经习惯了在 `<script>` 块中写 JavaScript，但希望你能了解这个差异：这仍然会有上下文切换的问题，你或者只写 HTML，或者只在 `<script>` 块中写 JavaScript。由 JavaScript 生成的 HTML 充满了问题：

- 你必须不断地考虑哪些字符需要转义以及如何转义。
- 使用 JavaScript 来生成那些自身包含 JavaScript 代码的 HTML 会很快让你抓狂。
- 你通常会失去编辑器的语法高亮显示和其他方便的语言特性。
- 很难发现格式不正确的 HTML。
- 很难直观地分析。
- 很难让别人读懂你的代码。

模板解决了在目标语言中编写代码的问题，同时也让插入动态数据成为了可能。用 Mustache 模板将之前的例子重写：

```
<h1>Much Better</h1>
<p>No <span class="code">document.write</span>here!</p>
<p>Today's date is {{today}}.</p>
```

现在我们要做的就是给 `{{today}}` 赋值，这就是模板语言的核心。

7.1 唯一一条绝对规则[1]

我并不是说一定不能在 JavaScript 中写 HTML，只是你应该尽量避免这么做。尤其要感谢像 jQuery 这样的类库，它让前端代码变得更加优美。例如，这样做我并不会有意见：

```
$('#error').html('Something <b>very bad</b> happened!');
```

然而，如果最终变成这样：

```
$('#error').html('<div class="error"><h3>Error</h3>' +
        '<p>Something <b><a href="/error-detail/' + errorNumber
        +'">very bad</a></b> ' +
        'happened. <a href="/try-again">Try again<a>, or ' +
        '<a href="/contact">contact support</a>.</p></div>');
```

也许就到了使用模板的时候了。重点是，我建议你在选择使用 HTML 字符串还是使用模板时做出最佳判断。我宁可使用模板，不管怎样，只有在某些最简单的情况下才会使用 JavaScript 生成 HTML。

7.2 选择模板引擎

在 Node 的世界里，有许多模板引擎可供选择，那么如何挑选呢？这是个复杂的问题，而

注 1：引用我的朋友和导师 Paul Inman 的话。

且大多取决于你的需求。下面是一些可供参考的准则。

- 性能
  显然，你希望模板引擎尽可能地快。任何时候你都不会希望网站被拖慢。

- 客户端、服务端或兼而有之？
  大多数（但不是所有）模板引擎都可用于客户端和服务器端。如果你需要在这两端都使用模板（并且你确实会这样做），我推荐你选择那些在两端都表现优秀的模板引擎。

- 抽象
  你想让代码更可读（例如，在普通 HTML 文本中使用大括号），或者你私下里厌恶 HTML 已久，希望有什么东西能把你从那些尖括号中拯救出来？模板（尤其是服务器端模板）在这里为你提供了一些选择。

这些只是在选择模板语言时较为突出的准则。如果你想要了解关于这一主题更详细的讨论，我强烈推荐你看看 Veena Basavaraj 的博客文章（http://engineering.linkedin.com/frontend/client-side-templating-throwdown-mustache-handlebars-dustjs-and-more），那里有她为 LinkedIn 选择模板引擎时的相关准则。

LinkedIn 选择的是 Dust（http://akdubya.github.io/dustjs/），但是 Handlebars（http://handlebarsjs.com/）也入围了，后者是我首选的模板引擎，并且接下来也要在本书中使用。

Express 允许使用你想要的任何一个模板引擎，所以，如果你并不喜欢 Handlebars，可以轻松地将它换掉。如果你想试试的话，可以使用这个有趣并且实用的工具：Template-Engine-Chooser（http://garann.github.io/template-chooser）。

7.3 Jade：不走寻常路

在大多数模板引擎还在以 HTML 为中心的时候，Jade 就以抽象 HTML 细节而引人注目。同样值得注意的是，Jade 是 TJ Holowaychuk 的设想，他也是为我们带来 Express 的人。那 Jade 和 Express 可以很好地结合也就不足为奇了。Jade 采用的方式是难能可贵的，其核心就是声称 HTML 是一种手写的模糊、枯燥的语言。让我们看看 Jade 模板是什么样的，同时也看看它输出的 HTML（取自 Jade 主页 http://jade-lang.com/，为适应此书格式稍作修改）：

```
doctype html                         <!DOCTYPE html>
html(lang="en")                      <html lang="en">
  head                               <head>
    title= pageTitle                 <title>Jade Demo</title>
    script.                          <script>
      if (foo) {                         if (foo) {
         bar(1 + 5)                          bar(1 + 5)
```

```
    }
  body

    h1 Jade
    #container
      if youAreUsingJade
        p You are amazing
      else
        p Get on it!
      p.
        Jade is a terse and
        simple templating
        language with a
        strong focus on
        performance and
        powerful features.
```

```
    }
</script>
<body>
<h1>Jade</h1>
<div id="container">

<p>You are amazing</p>


<p>
  Jade is a terse and
  simple templating
  language with a
  strong focus on
  performance and
  powerful features.
</p>
</body>
</html>
```

Jade 无疑是少打了很多字，因为不再有尖括号和结束标记。取而代之，它依赖缩进和一些常识性规则，从而更容易表达出自己想要的。Jade 具有一个额外的优势：理论上讲，当 HTML 自身发生改变时，你可以轻松地将 Jade 定位于 HTML 版本的最新版本，从而让你的内容更具“前瞻性”。

尽管我赞赏 Jade 的理念和优雅的执行，但是我发现，我并不想让 HTML 过于抽象。作为一个 Web 开发者，HTML 是核心，如果代价是尖括号从我的键盘上磨损掉，那也没关系。大部分前端开发人员的感受都如我所述，也许世界还没有准备好接受 Jade……

从这里开始，我们要与 Jade 分道扬镳，在本书中你不会再见到它。然而，如果抽象概念很吸引你，并且你确定在 Express 中使用 Jade 没有问题，还有很多资料可以帮助你。

7.4 Handlebars基础

Handlebars 是另一个流行的模板引擎 Mustache 的扩展。我推荐 Handlebars，是因为它简单的 JavaScript 集成（前端和后端）和容易掌握的语法。对我来说，它实现了所有的平衡，也是本书中要关注的。尽管我们正在讨论的概念是适用于其他模板的，但如果你觉得 Handlebars 不能激发你的想象力，可以去尝试其他不一样的模板引擎。

理解模板引擎的关键在于 context（上下文环境）。当你渲染一个模板时，便会传递给模板引擎一个对象，叫作上下文对象，它能让替换标识运行。

例如，如果上下文对象是 `{ name: 'Buttercup' }`，模板是 `<p>Hello, {{name}}!</p>`，则 `{{name}}` 会被 `Buttercup` 替换。如果向模板中传递 HTML 文本会发生什么呢？例如，上下文换成 `{ name: '<b>Buttercup</b>' }`，使用之前的模板得到的结果将是 `<p>Hello,<b&g`

t;Buttercup<b></p>，这或许并不是你想要的。要想解决这个问题，用三个大括号代替两个就可以了：{{{name}}}。

> 虽然我们已经确定了应当避免在 JavaScript 中编写 HTML 代码，但是使用三重大括号关闭 HTML 转义的功能具有一些重要用途。例如，如果用 WYSIWYG 编辑器建立了一个 CMS 系统，你大概会希望向视图层传递 HTML 文本是可行的。此外，能够脱离 HTML 转义渲染上下文属性对于布局和章节是很重要的，这一点我们不久就会了解到。

在图 7-1 中，我们可以看到 Handlebars 引擎是怎样使用上下文（用椭圆表示）结合模板渲染 HTML 的。

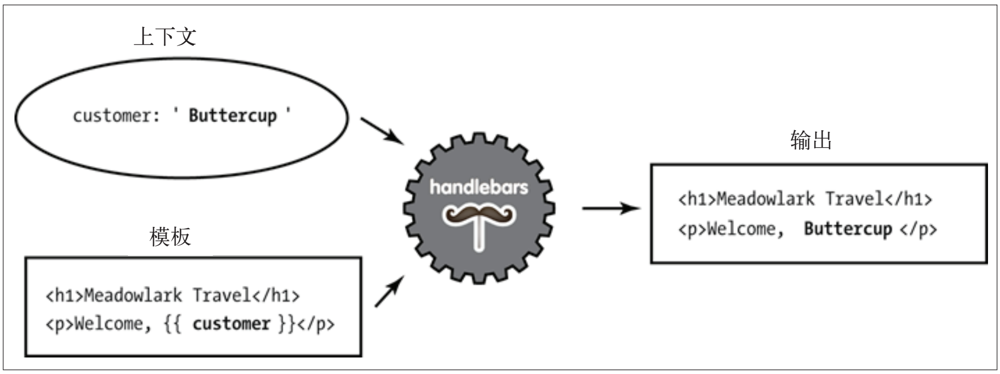


图 7-1　使用 Handlebars 渲染 HTML

7.4.1　注释

Handlebars 的注释看起来像 {{! comment goes here }}。懂得如何区分 Handlebars 注释和 HTML 注释很重要。示例如下：

```
{{! super-secret comment }}
<!-- not-so-secret comment -->
```

假设这是一个服务器端模板，上面的 super-secret comment 将不会被传递到浏览器，然而如果用户查看 HTML 源文件，下面的 not-so-secret comment 就会被看到。你应该会喜欢 Handlebars 注释那些需要显示实现细节的地方，或者是你不想暴露出来的其他任何东西。

7.4.2　块级表达式

当你考虑块级表达式（block）的时候，事情就开始变得复杂了。块级表达式提供了流程控制、条件执行和可扩展性。看一下下面的上下文对象：

```
{
	currency: {
		name: 'United States dollars',
		abbrev: 'USD',
	},
	tours: [
		{ name: 'Hood River', price: '$99.95' },
		{ name: 'Oregon Coast', price, '$159.95' },
	],
	specialsUrl: '/january-specials',
	currencies: [ 'USD', 'GBP', 'BTC' ],
}
```

现在让我们将上下文对象传递到如下模板：

```
<ul>
	{{#each tours}}
		{{! I'm in a new block...and the context has changed }}
		<li>
			{{name}} - {{price}}
			{{#if ../currencies}}
				({{../../currency.abbrev}})
			{{/if}}
		</li>
	{{/each}}
</ul>
{{#unless currencies}}
	<p>All prices in {{currency.name}}.</p>
{{/unless}}
{{#if specialsUrl}}
	{{! I'm in a new block...but the context hasn't changed (sortof) }}
	<p>Check out our <a href="{{specialsUrl}}">specials!</p>
{{else}}
	<p>Please check back often for specials.</p>
{{/if}}
<p>
	{{#each currencies}}
		<a href="#" class="currency">{{.}}</a>
	{{else}}
		Unfortunately, we currently only accept {{currency.name}}.
	{{/each}}
</p>
```

这个模板很复杂，所以让我们分解一下。它开始于 `each` 辅助方法，这使我们能够遍历一个数组。理解 `{{#each tours}}` 和 `{{/each tours}}` 之间的东西很重要，这涉及上下文切换。第一次循环，上下文变成了 `{ name: 'Hood River', price: '$99.95' }`，第二次则变成了 `{ name: 'Oregon Coast', price: '$159.95' }`。所以在这个块里面，我们可以看到 `{{name}}` 和 `{{price}}`。然而，如果你想访问 `currency` 对象，就得使用 `../` 来访问上一级上下文。

如果上下文属性本身就是一个对象，我们可以直截了当地访问它的属性，比如 `{{currency.name}}`。

`if` 辅助方法有些特殊，也有点让人困惑。在 Handlebars 中，所有的块都会改变上下文，所以在 `if` 块中，会产生一个新的上下文……而这刚好是上一级上下文的副本。换句话说，在 `if` 或 `else` 块中，上下文与上一级上下文是相同的。上述实现细节通常是显而易见的，但是当你在一个 `each` 循环中使用 `if` 块时就有必要细究一下了。在 `{{#each tours}}` 循环体中，可以使用 `../.` 访问上级上下文。不过，在 `{{#if ../currencies}}` 块中，又进入了一个新的上下文……所以要获得 `currency` 对象，就得使用 `../../.`。第一个 `../` 获得产品的上下文，第二个获得最外层的上下文。这就会产生很多混乱，最简单的权宜之计就是在 `each` 块中避免使用 `if` 块。

在 `if` 和 `each` 块中都有一个可选的 `else` 块（对于 `each`，如果数组中没有任何元素，`else` 块就会执行）。我们也用到了 `unless` 辅助方法，它基本上和 `if` 辅助方法是相反的：只有在参数为 `false` 时，它才会执行。

在这个模板中，最后要注意的一点是在 `{{#each currencies}}` 块中使用 `{{.}}`。`{{.}}` 指向当前上下文，在这个例子中，当前上下文只是我们想打印出来的数组中的一个字符串。

> 访问当前上下文还有另外一种独特的用法：它可以从当前上下文的属性中区分出辅助方法（我们很快就会学到）。例如，如果有一个辅助方法叫作 `foo`，在当前上下文中有一个属性也叫作 `foo`，则 `{{foo}}` 指向辅助方法，`{{./foo}}` 指向属性。

7.4.3　服务器端模板

服务器端模板会在 HTML 发送到客户端之前渲染它。服务器端模板与客户端模板不同，客户端模板能够被懂得如何查看 HTML 源文件的富有好奇心的用户看到，而你的用户将不会看到服务器端模板，或是用于最终生成 HTML 的上下文对象。

服务器端模板除了隐藏实现细节，还支持模板*缓存*，这对性能很重要。模板引擎会缓存已编译的模板（只有在模板发生改变的时候才会重新编译和重新缓存），这会改进模板视图的性能。默认情况下，视图缓存会在开发模式下禁用，在生产模式下启用。如果想显式地启用视图缓存，可以这样做：`app.set('view cache', true);`。

Express 支持 Jade、EJS 和 JSHTML。我们已经讨论过 Jade 了，而且我觉得 EJS 和 JSHTML 也不值得推荐（在我看来，在语法上做得还不够）。所以我们需要添加一个 node 包，让 Express 提供 Handlebars 支持。

```
npm install --save express3-handlebars
```

然后就可以在 Express 中引入：

```
var handlebars = require('express3-handlebars')
        .create({ defaultLayout: 'main' });
app.engine('handlebars', handlebars.engine);
app.set('view engine', 'handlebars');
```

> express3-handlebars 让 Handlebars 模板拥有了 .handlebars 扩展名。我已经习惯了，但是这对你来说太冗长了，你可以在创建 express3-handlebats 实例 require('express3-handlebars').create({ extname: '.hbs' }) 的时候，将扩展名改成同样常见的 .hbs。

7.4.4　视图和布局

视图通常表现为网站上的各个页面（它也可以表现为页面中 AJAX 局部加载的内容，或一封电子邮件，或页面上的任何东西）。默认情况下，Express 会在 views 子目录中查找视图。*布局*是一种特殊的视图，事实上，它是一个用于模板的模板。布局是必不可少的，因为站点的大部分（即使不是全部）页面都有几乎相同的布局。例如，页面中必须有一个 <html> 元素和一个 <title> 元素，它们通常都会加载相同的 CSS 文件，诸如此类。你不想为每个网页复制代码，于是这就需要用到布局。让我们看看基本的布局文件：

```
<!doctype>
<html>
<head>
        <title>Meadowlark Travel</title>
        <link rel="stylesheet" href="/css/main.css">
</head>
<body>
        {{{body}}}
</body>
</html>
```

请注意 <body> 标记内的文本：{{{body}}}。这样视图引擎就知道在哪里渲染你的内容了。一定要用三重大括号而不是两个，因为视图很可能包含 HTML，我们并不想让 Handlebars 试图去转义它。注意，在哪里放置 {{{body}}} 并没有限制。例如，你想用 Bootstrap 3 构建一个响应式布局，你或许想要把视图放进一个 <div> 容器里。此外，常见的网页元素，如页眉和页脚，通常也在布局中，而不在视图中。举例如下：

```
<!-- ... -->
<body>
        <div class="container">
                <header><h1>Meadowlark Travel</h1></header>
                {{{body}}}
                <footer>&copy; {{copyrightYear}} Meadowlark Travel</footer>
        </div>
</body>
```

图 7-2 展示了模板引擎是怎样结合视图、布局和上下文来完成渲染的。重要的是，此图解释了运行的顺序。视图首先被渲染，之后是布局。起初这看似是反常的：视图是在布局中渲染的，所以不应该是布局首先被渲染吗？虽然从技术上讲可以这么做，但是逆向运行是有优势的。特别是，它允许视图本身进一步自定义布局，这在我们讨论段落时会派上用场。

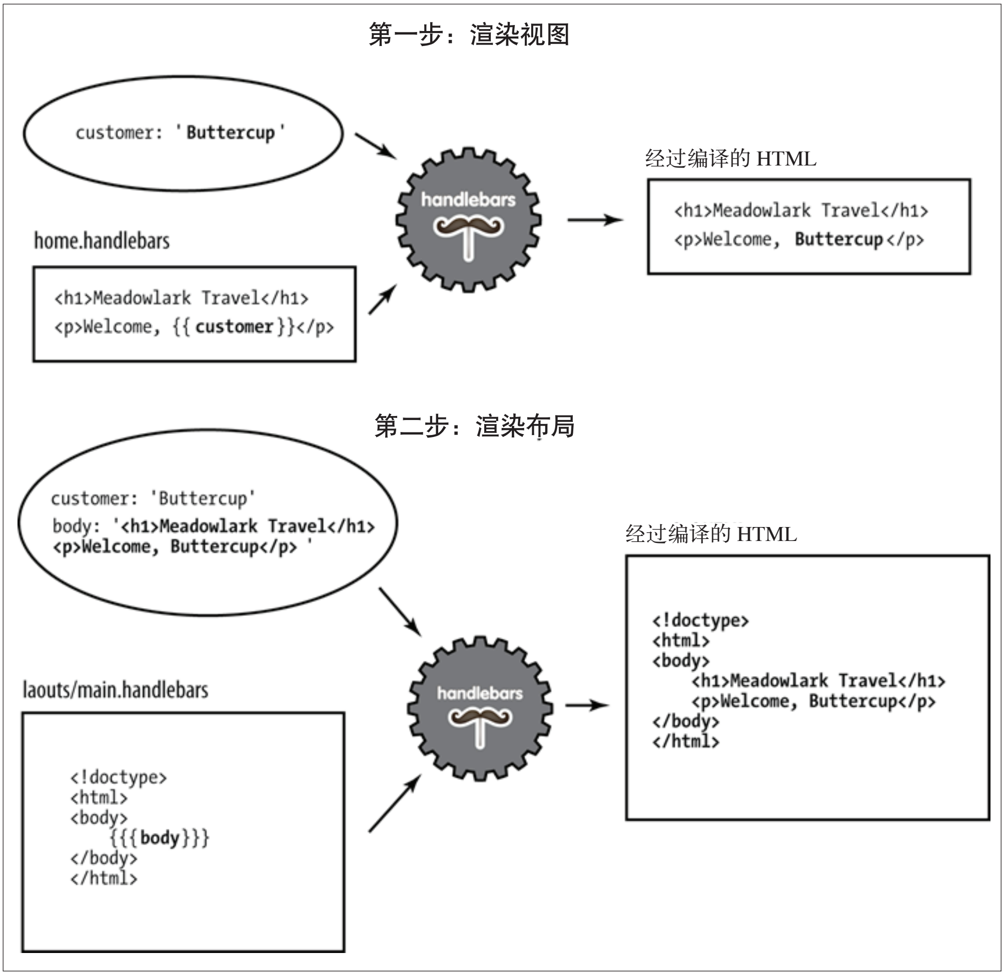


图 7-2　使用布局渲染视图

> 由于执行的顺序，你可以向视图中传递一个叫作 body 的属性，而且它会在视图中正确渲染。然而，当布局被渲染时，body 的值会被已渲染的视图覆盖。

7.4.5 在Express中使用（或不使用）布局

很有可能，大部分（即使不是全部）页面都采用相同的布局，所以在每次渲染视图的时候都为其制定一个布局是不合理的。你会注意到，当我们创建视图引擎时，会指定一个默认的布局：

```
var handlebars = require('express3-handlebars')
        .create({ defaultLayout: 'main' });
```

默认情况下，Express 会在 views 子目录中查找视图，在 views/layouts 下查找布局。所以如果有一个叫作 views/foo.handlebars 的视图，可以这样渲染它：

```
app.get('/foo', function(req, res){
        res.render('foo');
});
```

它会使用 views/layouts/main.handlebars 作为布局。如果你根本不想使用布局（这意味着在视图中你不得不拥有所有的样板文件），可以在上下文中指定 `layout: null`：

```
app.get('/foo', function(req, res){
        res.render('foo', { layout: null });
});
```

或者，如果你想使用一个不同的模板，可以指定模板名称：

```
app.get('/foo', function(req, res){
        res.render('foo', { layout: 'microsite' });
});
```

这样就会使用布局 `views/layouts/microsite.handlebars` 来渲染视图了。

需要注意的是，你拥有的模板越多，需要维护的基础 HTML 布局就越多。另一方面，如果你的页面在布局上有很大的不同，这也许是值得的。针对自己的项目，你必须找到一种平衡。

7.4.6 局部文件

很多时候，有些组成部分（在前端界通常称为“组件”）需要在不同的页面重复使用。使用模板来实现这一目标的唯一方法是使用*局部文件*（partial，如此命名是因为它们并不渲染整个视图或整个网页）。设想一下，如果有一个当前天气组件用来显示 Portland、Bend 和 Manzanita 三地的天气条件。我们希望这个组件可以被重复使用，这样就可以轻松地把它放在任何我们想让它出现的页面上，这就要用到局部文件。首先，创建一个局部文件，views/partials/weather.handlebars：

```
<div class="weatherWidget">
```

```
        {{#each partials.weather.locations}}
            <div class="location">
                <h3>{{name}}</h3>
                <a href="{{forecastUrl}}">
                    <img src="{{iconUrl}}" alt="{{weather}}">
                    {{weather}}, {{temp}}
                </a>
            </div>
        {{/each}}
        <small>Source: <a href="http://www.wunderground.com">Weather
            Underground</a></small>
</div>
```

请注意，我们使用 partials.weather 为开头来命名上下文。我们想在任何页面上使用局部文件，但上述做法实际上并不会将上下文传递给每一个视图，因此可以使用 res.locals（对于任何视图可用）。但是我们并不想让个别的视图干扰指定的上下文，于是将所有的局部文件上下文都放在 `partials` 对象中。

在第 19 章中，我们将会看到如何通过免费的 Weather Underground API 来获得当前天气信息。现在，我们要使用虚拟数据。在应用程序文件中，我们要创建一个方法来获取当前天气数据：

```
function getWeatherData(){
    return {
        locations: [
            {
                name: 'Portland',
                forecastUrl: 'http://www.wunderground.com/US/OR/Portland.html',
                iconUrl: 'http://icons-ak.wxug.com/i/c/k/cloudy.gif',
                weather: 'Overcast',
                temp: '54.1 F (12.3 C)',
            },
            {
                name: 'Bend',
                forecastUrl: 'http://www.wunderground.com/US/OR/Bend.html',
                iconUrl: 'http://icons-ak.wxug.com/i/c/k/partlycloudy.gif',
                weather: 'Partly Cloudy',
                temp: '55.0 F (12.8 C)',
            },
            {
                name: 'Manzanita',
                forecastUrl: 'http://www.wunderground.com/US/OR/Manzanita.html',
                iconUrl: 'http://icons-ak.wxug.com/i/c/k/rain.gif',
                weather: 'Light Rain',
                temp: '55.0 F (12.8 C)',
            },
        ],
    };
}
```

现在创建一个中间件给 `res.locals.partials` 对象添加这些数据（我们将在第 10 章详细学

习中间件）：

```
app.use(function(req, res, next){
        if(!res.locals.partials) res.locals.partials = {};
        res.locals.partials.weather = getWeatherData();
        next();
});
```

现在所有的东西都准备好了，我们所要做的就是在视图中使用这个局部文件。例如，为将我们的组件放在主页上，编辑 views/home.handlebars：

```
<h2>Welcome to Meadowlark Travel!</h2>
{{> weather}}
```

语法 {{> partial_name}} 可以让你在视图中包含一个局部文件。express3-handlebars 会在 views/partials 中寻找一个叫作 partial_name.handle-bars 的视图（或是 weather.handlebars，如上例）。

> express3-handlebars 支持子目录，所以如果你有大量的局部文件，可以将它们组织在一起。例如，你有一些社交媒体局部文件，可以将它们放在 views/partials/social 目录下面，然后使用 {{> social/facebook}}、{{> social/twitter}} 等来引入它们。

7.4.7　段落

我从微软的优秀模板引擎 Razor 中借鉴了段落（section）的概念。如果所有的视图在你的布局中都正好放在一个单独的元素里，布局会正常运转，但是当你的视图本身需要添加到布局的不同部分时会发生什么？一个常见的例子是，视图需要向 <head> 元素中添加一些东西，或是插入一段使用 jQuery 的 <script> 脚本（这意味着必须引入 jQuery，由于性能原因，有时在布局中这是最后才做的事）。

Handlebars 和 express3-handlebars 都没有针对于此的内置方法。幸运的是，Handlebars 的辅助方法让整件事情变得简单起来。当我们实例化 Handlebars 对象时，会添加一个叫作 section 的辅助方法：

```
var handlebars = require('express3-handlebars').create({
    defaultLayout:'main',
    helpers: {
        section: function(name, options){
            if(!this._sections) this._sections = {};
            this._sections[name] = options.fn(this);
            return null;
        }
    }
});
```

现在我们可以在视图中使用 section 辅助方法了。让我们创建一个视图（views/jquerytest.handlebars），在 <head> 中添加一些东西，并添加一段使用 jQuery 的脚本：

```
{{#section 'head'}}
        <!-- we want Google to ignore this page -->
        <meta name="robots" content="noindex">
{{/section}}

<h1>Test Page</h1>
<p>We're testing some jQuery stuff.</p>

{{#section 'jquery'}}
        <script>
                $('document').ready(function(){
                        $('h1').html('jQuery Works');
                });
        </script>
{{/section}}
```

现在在这个布局里，我们可以像放置 {{{body}}} 一样放置一个段落：

```
<!doctype html>
<html>
<head>
        <title>Meadowlark Travel</title>
        {{{_sections.head}}}
</head>
<body>
        {{{body}}}
        <script src="http://code.jquery.com/jquery-2.0.2.min.js"></script>
        {{{_sections.jquery}}}
</body>
</html>
```

7.4.8 完善你的模板

模板是网站的核心。一个好的模板结构将会为你节省开发时间，促进网站的一致性，还可以减少差异布局的数量。为了实现这些目标，你必须花费一些时间仔细构想你的模板。决定使用多少模板是一种艺术。一般来说，越少越好，但是有一种观点认为收益递减，这取决于页面的一致性。模板也是应对跨浏览器兼容问题和有效网页标准的第一道防线。它们应该由精通前端开发的人来精心编制和维护。从 HTML5 Boilerplate（http://html5boilerplate.com/）开始是个好的选择，尤其对于新手来说。在前面的示例中，我们使用了 HTML5 最小的模板来适应此书的格式，但是在实际项目中，要使用 HTML5 Boilerplate。

另一种流行的方式是使用第三方主题。像 Themeforest（http://themeforest.net/category/site-templates）和 WrapBootstrap（https://wrapbootstrap.com/）这样的网站有几百种 HTML5 即

用模板，它们可以用来开发你的第一个模板。使用第三方主题要从考虑主文件（通常是index.html）入手，将它重命名为 main.handlebars（也可以任意命名你的布局文件），将静态资源（CSS 样式文件、JavaScript 脚本、图片）放在公共目录下。然后，你需要编辑模板文件并指出在什么地方放置 {{{body}}} 表达式。根据你模板上的元素，你也许会想将其中一些放在局部文件中。一个非常好的例子就是“hero”（一种为了吸引用户眼球而设计的高高的横幅）。如果 hero 出现在每一个页面上（这可能是个糟糕的选择），你应该把它放在模板文件里。如果它只出现在一个页面里（通常是主页），那应该只把它放在那个视图里。如果它出现在几个（但不是全部）页面中，那么你可能需要考虑将它放在局部文件中。选择权在你，这展现了制作一个独特而又充满魅力网站的艺术性。

7.4.9　客户端Handlebars

当你想显示动态内容的时候，handlebars 的客户端模板就派上用场了。当然，AJAX 调用可以返回 HTML 片段，并将其原样插入 DOM 中，但是客户端 Handlebars 允许我们使用JSON 数据接收 AJAX 调用结果，并将其格式化以适应我们的网站。因此，在与第三方API（返回 JSON 数据，而不是适应你网站的格式化 HTML 文本）通信时尤其有用。

在客户端使用 Handlebars 之前，我们需要加载 Handlebars。我们既可以将 Handlebars 放在静态资源中引入，也可以使用一个 CDN。我们在 views/nursery-rhyme.handlebars 中使用第二种方法：

```
{{#section 'head'}}
        <script src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.3.0/
  handlebars.min.js"></script>
{{/section}}
```

现在需要找个地方放我们的模板了。一种方法是使用在 HTML 中已存在的元素，最好是一个隐藏的元素。你可以将它放在 `<head>` 中的 `<script>` 元素里。这看起来有点奇怪，但是运行良好：

```
{{#section 'head'}}
        <script src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.3.0/
handlebars.min.js"></script>

        <script id="nurseryRhymeTemplate" type="text/x-handlebars-template">
                Marry had a little <b>\{{animal}}</b>, its <b>\{{bodyPart}}</b>
                was <b>\{{adjective}}</b> as <b>\{{noun}}</b>.
        </script>
{{/section}}
```

请注意，我们必须转义至少一个大括号，否则，服务器端视图会尝试对其进行替换。

在使用模板之前，我们需要编译它：

```
{{#section 'jquery'}}
        $(document).ready(function(){
                var nurseryRhymeTemplate = Handlebars.compile(
                        $('#nurseryRhymeTemplate').html());
        });
{{/section}}
```

我们需要一个放置已渲染模板的地方。出于测试的目的，我们添加两个按钮，一个通过 JavaScript 来直接渲染，另一个通过 AJAX 调用来渲染：

```
<div id="nurseryRhyme">Click a button....</div>
<hr>
<button id="btnNurseryRhyme">Generate nursery rhyme</button>
<button id="btnNurseryRhymeAjax">Generate nursery rhyme from AJAX</button>
```

最后是渲染模板的代码：

```
{{#section 'jquery'}}
        <script>
                $(document).ready(function(){

                        var nurseryRhymeTemplate = Handlebars.compile(
                                $('#nurseryRhymeTemplate').html());

                        var $nurseryRhyme = $('#nurseryRhyme');

                        $('#btnNurseryRhyme').on('click', function(evt){
                                evt.preventDefault();
                                $nurseryRhyme.html(nurseryRhymeTemplate({
                                        animal: 'basilisk',
                                        bodyPart: 'tail',
                                        adjective: 'sharp',
                                        noun: 'a needle'
                                }));
                        });

                        $('#btnNurseryRhymeAjax').on('click', function(evt){
                                evt.preventDefault();
                                $.ajax('/data/nursery-rhyme', {
                                        success: function(data){
                                                $nurseryRhyme.html(
                                                        nurseryRhymeTemplate(data))
                                        }
                                });
                        });
                });
        </script>
{{/section}}
```

针对 nursery rhyme 页和 AJAX 调用的路由处理程序：

```
app.get('/nursery-rhyme', function(req, res){
        res.render('nursery-rhyme');
```

```
    });
    app.get('/data/nursery-rhyme', function(req, res){
            res.json({
                    animal: 'squirrel',
                    bodyPart: 'tail',
                    adjective: 'bushy',
                    noun: 'heck',
            });
    });
```

从本质上讲，`Handlebars.compile` 接收一个模板，返回一个方法。这个方法接收一个上下文对象，返回一个已渲染字符串。所以一旦我们编译了模板，就可以像调用方法函数一样重用模板渲染。

7.5 小结

我们已经看到了模板是如何让你的代码易写、易读、易维护的。因为模板，我们不需要在 JavaScript 中痛苦地拼凑 HTML 字符串了。我们可以在喜欢的编辑器中写 HTML，并且可以使用一个小巧易读的模板语言使其动态化。

第 8 章

表单处理

从用户那里收集信息的常用方法就是使用 HTML 表单。无论是使用浏览器提交表单，还是使用 AJAX 提交，或是运用精巧的前端控件，底层机制通常仍旧是 HTML 表单。在这一章，我们将讨论不同的表单处理方法、表单验证和文件上传。

8.1　向服务器发送客户端数据

大体上讲，向服务器发送客户端数据有两种方式：查询字符串和请求正文。通常，如果是使用查询字符串，就发起了一个 `GET` 请求；如果是使用请求正文，就发起了一个 `POST` 请求（如果你反过来做，HTTP 协议并不会阻止你，但这是没有必要的：最好在这里坚持标准实践）。

有一种普遍的误解是 `POST` 请求是安全的，而 `GET` 请求不安全。事实上如果使用 HTTPS 协议，两者都是安全的；如果不使用，则都不安全。如果不使用 HTTPS 协议，入侵者会像查看 `GET` 请求的查询字符串一样，轻松查看 `POST` 请求的报文数据。然而，如果你使用 `GET` 请求，用户会在查询字符串中看到所有的输入数据（包括隐藏域），这是丑陋而且凌乱的。此外，浏览器会限制查询字符串的长度（对请求正文没有长度限制）。基于这些原因，一般推荐使用 `POST` 进行表单提交。

8.2　HTML表单

这本书侧重于服务器端，但重要的是需要了解一些构建 HTML 表单的基础知识。下面是一个简单的例子：

```
<form action="/process" method="POST">
    <input type="hidden" name="hush" val="hidden, but not secret!">
    <div>
        <label for="fieldColor">Your favorite color: </label>
        <input type="text" id="fieldColor" name="color">
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</form>
```

请注意，在 `<form>` 标记中提交方法被明确地指定为 POST：如果不这么做，默认进行 GET 提交。action 的值被指定为用于接收表单数据的 URL。如果你忽略这个值，表单会提交到它被加载进来时的同一 URL。我建议你始终都为 action 提供一个有效值，即使是使用 AJAX 提交（这会防止你丢失数据，详见第 22 章）。

从服务器的角度来看，最重要的属性是 `<input>` 域中的 name 属性，这样服务器才能识别字段。name 属性与 id 属性是截然不同的，后者只适用于样式和前端功能（它不会发送到服务器端），理解这一点非常重要。

注意隐藏域：它不会呈现在浏览器中。但是，你不能使用它存放秘密和敏感信息：用户只要查看页面源文件，隐藏域就会暴露出来。

HTML 并不会限制在同一个页面上有多个表单（遗憾的是有些早期服务器框架有限制，比如 ASP）。[1] 我建议你保持表达逻辑上的一致性：一个表单应该只包含你想要提交的字段（可选的 / 空字段也可以）。如果一个页面上有两个不同的 action，请使用两个不同的表单。例如，在一个页面上一个表单用于网站搜索，另一个表单用于登录获得电子简讯。只用一个大表单是可行的，可以根据用户点击的按钮判断采用哪个 action，但是这会让人头疼，而且通常对于残疾人是不友好的（由于无障碍浏览器呈现表单的方式）。

当用户提交表单时，/process URL 被请求，字段值在请求正文中被传输到服务器。

8.3 编码

当表单被提交（通过浏览器或 AJAX）时，某种程度上它必须被编码。如果不明确地指定编码，则默认为 application/x-wwwform-urlencoded（这只是一个冗长的用于“URL 编码”的媒体类型）。它是受 Express 支持的基本、易用的编码。

如果你需要上传文件，事情就开始变得复杂起来。使用 URL 编码很难发送文件，所以你不得不使用 multipart/form-data 编码类型，这并不直接由 Express 处理（事实上，Express

注 1：非常老的浏览器在处理多表单时可能会出现问题，所以如果你的目标是尽可能实现最大程度的兼容，那么可能需要考虑每页只使用一个表单。

仍然支持这种编码，但是在 Express 的下一个版本它会被移除，并且它也并不被建议使用。我们不久将会讨论它的替代品）。

8.4 处理表单的不同方式

如果不使用 AJAX，你唯一的选择是用浏览器提交表单，这会重新加载页面。然而，如何重新加载页面由你来决定。处理表单时有两件事需要考虑：处理表单是哪个路径（action），以及向浏览器发出怎样的响应。

如果你的表单使用的是 `method="POST"`（推荐使用），那么展现表单和处理表单通常使用相同的路径：这样可以区分开来，因为前者是一个 `GET` 请求，而后者是一个 `POST` 请求。如果采用这种方法，就可以省略表单上的 `action` 属性。

另一种选择是使用一个单独的路径处理表单。例如，如果使用路径 /contact 触发页面，你可以使用路径 /process-contact 来处理表单（通过指定 `action="/process-contact"`）。如果采用这种方法，你可以选择通过 `GET` 来提交表单（我不建议你这样做，因为这样会不必要地在 URL 中暴露你的表单域信息）。如果有多个 URL 使用了相同的提交方法，这种方法可能是首选（例如，你可能在站点的多个页面上有电子邮件登录框）。

无论使用什么路径来处理表单，必须决定如何响应浏览器。下面是你的选项。

- 直接响应HTML

  处理表单之后，可以直接向浏览器返回 HTML（例如，一个视图）。如果用户尝试重新加载页面，这种方法就会产生警告，并且会影响书签和后退按钮。基于这些原因，我们不推荐这种方法。

- 302重定向

  虽然这是一种常见的方法，但这是对响应代码 302（已找到）本义的滥用。HTTP 1.1 增加了响应代码 303（请参阅其他），一种更合适的代码。除非你有理由让浏览器回到 1996 年，否则你应该改用 303。

- 303重定向

  HTTP 1.1 添加了响应代码 303（请参阅其他）用来解决 302 重定向的滥用。HTTP 规范明确地表明浏览器 303 重定向后，无论之前是什么方法，都应该使用 `GET` 请求。这是用于响应表单提交请求的推荐方法。

由于推荐你通过 303 重定向来响应表单提交，接下来的问题是：“重定向指向哪里？”答案是，随你便。下面是一些常用的方法。

- 重定向到专用的成功/失败页面

  这种方法需要为适当的成功或失败消息提供 URL。例如，如果一个用户通过促销邮件注册，但是有一个数据库错误，你可能希望重定向到 /error/database。如果用户的电子邮件地址是无效的，可以重定向到 /error/invalid-email。如果一切顺利，可以重定向到 /promo-email/thank-you。这种方法的一个优点是便于分析：访问 /promo-email/thank-you 页面的人数应该和登录促销邮件的人数大致相关。而且这种方法也很容易实现。然而它还有一些缺点。这意味着你必须针对每一种可能性来分配 URL，这也意味着页面设计、编写复制和维护。另一个缺点是用户体验欠佳：用户喜欢被感谢，但是他们不得不导航到之前的页面或接下来要去的页面。这是现在我们要使用的方法。在第 9 章将使用 flash 消息（不要和 Adobe Flash 混淆）。

- 运用flash消息重定向到原位置

  由于有许多小表单分散在整个站点中（例如，电子邮件登录），最好的用户体验是不干扰用户的导航流。也就是说，需要一个不用离开当前页面就能提交表单的方法。当然，要做到这一点，可以用 AJAX，但是如果你不想用 AJAX（或者你希望备用机制能够提供一个好的用户体验），可以重定向回用户之前浏览的页面。最简单的方法是在表单中使用一个隐藏域来存放当前 URL。因为你想有一种反馈，表明用户的提交信息已收到，所以你可以使用 flash 消息。

- 运用flash消息重定向到新位置

  大型表单通常都会有自己的页面，一旦提交就没有必要停留在这个页面上了。在这种情况下，你就要考虑一下用户接下来想去哪儿，并相应地进行重定向。例如，如果你构建一个管理界面，有一个表单用来创建旅行计划，大概能够很合理地预期用户希望在提交表单后跳转到管理页，并且列出所有的旅行计划清单。不管怎样，你应该仍旧采用 flash 消息为用户提供提交结果的反馈。

如果使用 AJAX，我推荐你使用专门的 URL。你可能想在 AJAX 处理器前加一个前缀（比如 /ajax/enter），但是我不鼓励采用这种方法，因为它把实现细节附加在 URL 上。而且，很快我们会看到，作为故障保障，AJAX 处理器应该处理常规的浏览器提交。

8.5 Express表单处理

如果使用 `GET` 进行表单处理，表单域在 `req.query` 对象中。例如，如果有一个名称属性为 `email` 的 HTML 输入字段，它的值会以 `req.query.email` 的形式传递到处理程序。关于这个方法真的不必多说，它就是这么简单。

如果使用 `POST`（推荐使用的），需要引入中间件来解析 URL 编码体。首先，安装 `body-parser` 中间件（`npm install --save body-parser`），然后引入：

```
app.use(require('body-parser')());
```

> 有时，你会发现有些地方不鼓励使用 express.bodyParser，并且理由充分。然而，这个问题在 Epress 4.0 中消失了，body-parser 中间件是安全的并且推荐使用。

一旦引入了 body-parser，你会发现 req.body 变为可用，这样所有的表单字段将可用。注意一点，req.body 并不阻止你使用查询字符串。让我们继续，在草地鹨旅行社中添加一个表单，让用户注册一个邮件列表。为了演示，我们将使用查询字符串、一个隐藏字段以及可视字段，详见 /views/newsletter.handlebars：

```
<h2>Sign up for our newsletter to receive news and specials!</h2>
<form class="form-horizontal" role="form"
        action="/process?form=newsletter" method="POST">
    <input type="hidden" name="_csrf" value="{{csrf}}">
    <div class="form-group">
        <label for="fieldName" class="col-sm-2 control-label">Name</label>
        <div class="col-sm-4">
            <input type="text" class="form-control"
            id="fieldName" name="name">
        </div>
    </div>
    <div class="form-group">
        <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
        <div class="col-sm-4">
            <input type="email" class="form-control" required
                id="fieldName" name="email">
        </div>
    </div>
    <div class="form-group">
        <div class="col-sm-offset-2 col-sm-4">
            <button type="submit" class="btn btn-default">Register</button>
        </div>
    </div>
</form>
```

注意，我们使用了 Twitter Bootstrap 样式，这也将贯穿本书其余部分。如果你不熟悉 Bootstrap，可能想参考 Twitter Bootstrap 文档（http://getbootstrap.com）。接下来看看示例 8-1。

示例 8-1　应用文件

```
app.use(require('body-parser')());

app.get('/newsletter', function(req, res){
    // 我们会在后面学到 CSRF……目前，只提供一个虚拟值
    res.render('newsletter', { csrf: 'CSRF token goes here' });
});
```

```
app.post('/process', function(req, res){
    console.log('Form (from querystring): ' + req.query.form);
    console.log('CSRF token (from hidden form field): ' + req.body._csrf);
    console.log('Name (from visible form field): ' + req.body.name);
    console.log('Email (from visible form field): ' + req.body.email);
    res.redirect(303, '/thank-you');
});
```

这就是所有的了。请注意，在处理程序中，我们将重定向到“thank you”视图。我们可以在此渲染视图，但是如果这样做，访问者的浏览器地址栏仍旧是 /process，这可能会令人困惑。发起一个重定向可以解决这个问题。

> 在这种情况下使用 303（或 302）重定向，而不是 301 重定向，这一点非常重要。301 重定向是“永久”的，意味着浏览器会缓存重定向目标。如果使用 301 重定向并且试图第二次提交表单，浏览器会绕过整个 `/process` 处理程序直接进入 `/thank you` 页面，因为它正确地认为重定向是永久性的。另一方面，303 重定向告诉浏览器“是的，你的请求有效，可以在这里找到响应”，并且不会缓存重定向目标。

8.6 处理AJAX表单

用 Express 处理 AJAX 表单非常简单；甚至可以使用相同的处理程序来处理 AJAX 请求和常规的浏览器回退。参考示例 8-2 和示例 8-3。

示例 8-2　HTML 文件（/views/newsletter.handlebars）

```
<div class="formContainer">
    <form class="form-horizontal newsletterForm" role="form"
            action="/process?form=newsletter" method="POST">
        <input type="hidden" name="_csrf" value="{{csrf}}">
        <div class="form-group">
            <label for="fieldName" class="col-sm-2 control-label">Name</label>
            <div class="col-sm-4">
                <input type="text" class="form-control"
                id="fieldName" name="name">
            </div>
        </div>
        <div class="form-group">
            <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
            <div class="col-sm-4">
                <input type="email" class="form-control" required
                    id="fieldName" name="email">
            </div>
        </div>
        <div class="form-group">
            <div class="col-sm-offset-2 col-sm-4">
                <button type="submit" class="btn btn-default">Register</button>
            </div>
```

```
            </div>
        </form>
    </div>
    {{#section 'jquery'}}
        <script>
            $(document).ready(function(){
                $('.newsletterForm').on('submit', function(evt){
                    evt.preventDefault();
                    var action = $(this).attr('action');
                    var $container = $(this).closest('.formContainer');
                    $.ajax({
                        url: action,
                        type: 'POST',
                        success: function(data){
                            if(data.success){
                                $container.html('<h2>Thank you!</h2>');
                            } else {
                                $container.html('There was a problem.');
                            }
                        },
                        error: function(){
                            $container.html('There was a problem.');
                        }
                    });
                });
            });
        </script>
    {{/section}}
```

示例 8-3　应用程序文件

```
    app.post('/process', function(req, res){
        if(req.xhr || req.accepts('json,html')==='json'){
            // 如果发生错误，应该发送 { error: 'error description' }
            res.send({ success: true });
        } else {
            // 如果发生错误，应该重定向到错误页面
            res.redirect(303, '/thank-you');
        }
    });
```

Express 提供了两个方便的属性：`req.xhr` 和 `req.accepts`。如果是 AJAX 请求（XHR 是 XML HTTP 请求的简称，AJAX 依赖于 XHR），`req.xhr` 值为 `true`。`req.accepts` 试图确定返回的最合适的响应类型。在此例中，`req.accepts('json,html')` 询问最佳返回格式是 JSON 还是 HTML：这可以根据 `Accepts` HTTP 头信息推断出来，它是浏览器提供的可读的、有序的响应类型列表。如果是一个 AJAX 请求，或者 User-Agent 明确要求 JSON 优先于 HTML，那么就会返回合适的 JSON 数据；否则，返回一个重定向。

在这个函数里可以做任何处理：通常会将数据保存到数据库。如果出现问题，则返回一个 `err` 属性（而不是 `success`）的 JSON 对象，或者重定向到一个错误页面（如果不是 AJAX 请求）。

在此例中，我们假设所有 AJAX 请求的是 JSON 数据，但是并没有要求 AJAX 通信必须使用 JSON（事实上，“X”在 AJAX 中代表 XML）。这个方法是 jQuery 友好的，因为通常 jQuery 假定所有数据都是 JSON 格式的。如果想让 AJAX 处理程序通用，或者知道 AJAX 请求使用 JSON 之外的东西，你应该根据 `Accepts` 头信息（可以根据 `req.accepts` 辅助方法轻松访问）返回一个适当的响应。如果响应完全基于 `Accepts` 头信息，你或许想看看 `c`，这是一个可以根据客户端预期轻松做出适当响应的简便方法。如果这样做，必须保证用 jQuery 发起 AJAX 请求时设置 `dataType` 和 `accepts` 属性。

8.7 文件上传

我们已经提到过，文件上传会带来一系列的并发症。幸好，有一些很棒的项目，可以让文件处理变成小菜一碟。

一般，文件上传可以使用 Connect 的内置中间件 `multipart` 来处理。但是，这个中间件已经从 Connect 中移除了，一旦 Express 更新了对 Connect 的依赖项，它也将从 Express 中消失，所以我强烈建议你不要使用这个中间件。

对于复合表单处理，目前有两种流行而健壮的选择：Busboy 和 Formidable。我发现 Formidable 要稍微简单一些，因为它有一个方便的回调方法，能够提供包含字段和文件信息的对象。对于 Busboy 而言，你必须对每一个字段和文件事件进行监听。因此我们会使用 Formidable 进行讲解。

虽然可以利用 XMLHttpRequest Level 2 的 `FormData` 接口（https://developer.mozilla.org/en-US/docs/Web/API/FormData）使用 AJAX 进行文件上传，但它只支持现代浏览器并且需要一些 jQuery 使用经验。后面我们会讨论 AJAX 的一个替代品。

让我们为草地鹨旅行社的旅行摄影比赛创建一个文件上传表单（views/contest/vacation-photo.handlebars）：

```
<form class="form-horizontal" role="form"
        enctype="multipart/form-data" method="POST"
        action="/contest/vacation-photo/{year}/{month}">
    <div class="form-group">
        <label for="fieldName" class="col-sm-2 control-label">Name</label>
        <div class="col-sm-4">
            <input type="text" class="form-control"
            id="fieldName" name="name">
        </div>
    </div>
    <div class="form-group">
```

```
            <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
            <div class="col-sm-4">
                <input type="email" class="form-control" required
                    id="fieldName" name="email">
            </div>
        </div>
        <div class="form-group">
            <label for="fieldPhoto" class="col-sm-2 control-label">Vacation photo
            </label>
            <div class="col-sm-4">
                <input type="file" class="form-control" required accept="image/*"
                    id="fieldPhoto" name="photo">
            </div>
        </div>
        <div class="form-group">
            <div class="col-sm-offset-2 col-sm-4">
                <button type="submit" class="btn btn-primary">Submit</button>
            </div>
        </div>
    </form>
```

注意，我们必须指定 `enctype="multipart/form-data"` 来启用文件上传。我们也可以通过 `accept` 属性来限制上传文件的类型（这是可选的）。

现在安装 Formidable（`npm install --save formidable`）并创建一下路由处理程序：

```
var formidable = require('formidable');

app.get('/contest/vacation-photo',function(req,res){
    var now = new Date();
    res.render('contest/vacation-photo',{
        year: now.getFullYear(),month: now.getMont()
    });
});

app.post('/contest/vacation-photo/:year/:month', function(req, res){
    var form = new formidable.IncomingForm();
    form.parse(req, function(err, fields, files){
        if(err) return res.redirect(303, '/error');
        console.log('received fields:');
        console.log(fields);
        console.log('received files:');
        console.log(files);
        res.redirect(303, '/thank-you');
    });
});
```

（year 和 month 被指定为路由参数，详见第 14 章）。继续运行，检查控制台日志。你会发现表单字段如你预期的那样：是一个有字段名称属性的对象。文件对象包含更多的数据，但这是相对简单的。对于每一个上传的文件，你会看到属性有文件大小、上传路径（通常是在临时目录中的一个随机名字），还有用户上传此文件的原始名字（文件名，而不是整

个路径，出于安全隐私考虑）。

接下来如何处理这个文件就取决于你了：可以将它保存到数据库，将其复制到更持久的位置，或者上传到云端文件存储系统。记住，如果你基于本地存储保存文件，应用程序不能很好地扩展，基于云端存储是一个更好的选择。在第 13 章我们会回顾这个例子。

8.8 jQuery文件上传

如果你想为用户提供真正别出心裁的文件上传，可拖拽，可以看到上传文件缩略图，并查看进度条，那我向你推荐 Sebastian Tschan 的 jQuery File Upload（http://blueimp.github.io/jQuery-File-Upload）。

设置 jQuery 文件上传并不是闲庭信步。幸好，有一个 npm 包能够帮助你在服务器端快刀斩乱麻。前端脚本是另一回事。jQuery File Upload 包使用 jQuery UI 和 Bootstrap，看起来相当便于使用。如果你想对它进行定制，那么就要做很多工作了。

要显示文件缩略图，`jquery-file-upload-middleware` 使用 ImageMagick（http://www.imagemagick.org），这是一个老牌儿的图像处理库。选择它意味着你的应用依赖于 ImageMagick，根据你主机环境的不同可能会导致一些不同的问题。在 Ubuntu 和 Debian 系统中，你可以使用 `apt-get install imagemagick` 安装 ImageMagick；在 OS X 中，你可以使用 `brew install imagemagick` 来安装。对于其他操作系统，请参考 ImageMagick 文档（http://www.imagemagick.org/script/binary-releases.php）。

让我们先从服务端设置。首先，安装 `jquery-file-upload-middleware` 包（`npm install --save jquery-file-upload-middleware`），然后在你的应用文件中添加以下代码：

```
var jqupload = require('jquery-file-upload-middleware');

app.use('/upload', function(req, res, next){
    var now = Date.now();
    jqupload.fileHandler({
        uploadDir: function(){
            return __dirname + '/public/uploads/' + now;
        },
        uploadUrl: function(){
            return '/uploads/' + now;
        },
    })(req, res, next);
});
```

如果你看看文档，会在“更复杂的示例”下面看到类似的例子。除非你为所有访问者提供一个共用的文件上传区域，否则你可能要将上传文件区分开来。简单的方法是创建一个时间戳目录来存储文件。更实际的做法是使用用户 ID 或其他唯一 ID 来创建子目录。例如，如果实现一个支持文件共享的聊天程序，你可能会使用聊天室的 ID。

请注意，我们将 jQuery File Upload 中间件挂载在 /upload 前缀上。你可以在这里使用任何前缀，但是确保该前缀不用于其他路由或中间件，不然会干扰文件上传操作。

接下来是文件上传的视图，你可以直接复制演示上传代码：你可以在 project’s GitHub 页面（https://github.com/blueimp/jQuery-File-Upload/releases）上传最新项目包。不可避免，程序包里有很多你不需要的东西，如 PHP 脚本和其他实现示例，你可以随便删除。大部分的文件应该放在公共目录中（这样可以提供静态服务），但是 HTML 文件需要复制到视图目录中。

如果你只想要一个可构建的最小示例，需要如下脚本：`js/vendor/jquery.ui.widget.js`、`js/jquery.iframe-transport.js` 和 `js/jquery.fileupload.js`。很显然，你也需要 jQuery。为了整洁，我一般喜欢把这些脚本放在 public/vendor/jqfu 目录下。在这个最小实现中，我们将 `<input type="file">` 元素放在 `<span>` 中，还有一个 `<div>` 用来列出所有已上传文件：

```
<span class="btn btn-default btn-file">
    Upload
    <input type="file" class="form-control" required accept="image/*"
        id="fieldPhoto" data-url="/upload" multiple name="photo">
</span>
<div id="uploads"></div>
```

然后我们加上 jQuery File Upload：

```
{{#section 'jquery'}}
    <script src="/vendor/jqfu/js/vendor/jquery.ui.widget.js"></script>
    <script src="/vendor/jqfu/js/jquery.iframe-transport.js"></script>
    <script src="/vendor/jqfu/js/jquery.fileupload.js"></script>
    <script>
        $(document).ready(function(){

            $('#fieldPhoto').fileupload({
                dataType: 'json',
                done: function(e, data){
                    $.each(data.result.files, function(index, file){
                        $('#fileUploads').append($('<div class="upload">' +
                             '<span class="glyphicon glyphicon-ok"></span>' +
                             ' ' + file.originalName + '</div>'));
                    });
                }
            });
        });
    </script>
{{/section}}
```

为上传按钮添加 CSS 动态样式：

```
.btn-file {
    position: relative;
    overflow: hidden;
```

```
    }
    .btn-file input[type=file] {
        position: absolute;
        top: 0;
        right: 0;
        min-width: 100%;
        min-height: 100%;
        font-size: 999px;
        text-align: right;
        filter: alpha(opacity=0);
        opacity: 0;
        outline: none;
        background: white;
        cursor: inherit;
        display: block;
    }
```

注意，`<input>` 标签里的 `data-url` 属性必须和用于中间件的路由前缀相匹配。在这个简单示例中，当一个文件上传完成后，一个 `<div class="upload">` 元素会附加到之前的 `<div id="uploads">` 下面。这个列表只显示文件名和大小，不提供删除、运行或者缩略图功能。但这是一个好的开始。定制 jQuery File Upload 演示程序会让人望而生畏，如果你的视角完全不同，从最小程序开始逐渐向上构建，而不是从演示和定制开始，可能会更简单。不管怎样，你会在 jQuery File Upload 文档网页（https://github.com/blueimp/jQuery-File-Upload/wiki）找到你想要的资源。

简单起见，草地鹨旅行社示例不会继续使用 jQuery File Upload，但是如果你希望看到这种方法的实现，请在资源库中参阅 `jquery-file-upload-example` 分支。

第 9 章

Cookie与会话

HTTP 是无状态协议。这就是说，当你在浏览器中加载页面，然后转到同一网站的另一页面时，服务器和浏览器都没有任何内在的方法可以认识到，这是同一浏览器访问同一网站。换一种说法，Web 工作的方式就是在每个 HTTP 请求中都要包含所有必要的信息，服务器才能满足这个请求。

尽管这是个问题，如果故事到这里就结束，我们将永远无法“登录”。流媒体也无法工作。网站不能记忆你从一个页面到下一个页面的喜好。所以我们需要用某种办法在 HTTP 上建立状态，于是便有了 cookie 和会话。

不幸的是，cookie 的名声并不好，因为人们用它做了些邪恶的事情。之所以说不幸，是因为 cookie 对“现代 Web”的功能真的至关重要（尽管 HTML5 已经引入了一些新特性，比如本地存储，它可以发挥相同的作用）。

cookie 的想法很简单：服务器发送一点信息，浏览器在一段可配置的时期内保存它。发送哪些信息确实是由服务器来决定：通常只是一个唯一 ID 号，标识特定浏览器，从而维持一个有状态的假象。

关于 cookie，有些重要的事情需要你了解：

- *cookie对用户来说不是加密的*
  服务器向客户端发送的所有 cookie 都能被客户端查看。你可以向客户端发送一些加密过的信息以保护其中的内容，但几乎不会有这种需求（至少在你不做坏事时是这样的）。我们会稍微讨论一下签名 cookie，它可以混淆 cookie 中的内容，但对于窥探者来说这

绝没有加密那样的安全性。

- *用户可以删除或禁用cookie*

  用户对 cookie 有绝对的控制权，并且浏览器支持批量或单个删除 cookie。除非你图谋不轨，否则用户没理由去删它，但在测试过程中有这种需求。用户也可以禁用 cookie，但这更容易造成问题，因为只有最简单的 Web 应用程序才不需要依赖 cookie。

- *一般的cookie可以被篡改*

  不管浏览器什么时候发起一个跟 cookie 关联的请求，只要你盲目地相信 cookie 中的内容，都有可能会受到攻击。比如说，有些极其愚蠢的人会执行 cookie 中的代码。要确保 cookie 不被篡改，请使用签名 cookie。

- *cookie可以用于攻击*

  这几年出现了一种叫作跨站脚本攻击（XSS）的攻击方式。XSS 攻击中有一种技术就涉及用恶意的 JavaScript 修改 cookie 中的内容。所以不要轻易相信返回到你的服务器的 cookie 内容。用签名 cookie 会有帮助（不管是用户修改的还是恶意 JavaScript 修改的，这些篡改都会在签名 cookie 中留下明显的痕迹），并且还可以设定选项指明 cookie 只能由服务器修改。这些 cookie 的用途会受限，但它们肯定更安全。

- *如果你滥用cookie，用户会注意到*

  如果你在用户的电脑上设了很多 cookie，或者存了很多数据，这可能会惹恼用户，所以你应该避免出现这种情况。尽量把对 cookie 的使用限制在最小范围内。

- *如果可以选择，会话要优于cookie*

  大多数情况下，你可以用*会话*维持状态，一般来说这样做是明智的。并且会话更容易，你不用担心会滥用用户的存储，而且也更安全。当然，会话要依赖 cookie，但如果你使用会话，Express 会帮你做很多工作。

> cookie 不是魔法。当服务器希望客户端保存一个 cookie 时，它会发送一个响应头 `Set-Cookie`，其中包含名称 / 值对。当客户端向服务器发送含有 cookie 的请求时，它会发送多个请求头 `Cookie`，其中包含这些 cookie 的值。

9.1 凭证的外化

为了保证 cookie 的安全，必须有一个 *cookie 秘钥*。cookie 秘钥是一个字符串，服务器知道它是什么，它会在 cookie 发送到客户端之前对 cookie 加密。这是一个不需要记住的密码，所以可以是随机字符串。我一般用一个随机密码生成器（受 xkcd 启发，http://preshing.com/20110811/xkcd-password-generator）来生成 cookie 秘钥。

外化第三方凭证是一种常见的做法，比如 cookie 秘钥、数据库密码和 API 令牌（Twitter、Facebook 等）。这不仅易于维护（容易找到和更新凭证），还可以让你的版本控制系统忽略这些凭证文件。这对放在 GitHub 或其他开源源码控制库上的开源代码库尤其重要。

因此我们准备将凭证外化在一个 JavaScript 文件中（用 JSON 或 XML 也行，但我觉得 JavaScript 最容易）。创建文件 credentials.js：

```
module.exports = {
    cookieSecret: ' 把你的 cookie 秘钥放在这里 ',
};
```

现在，为了防止我们不慎把这个文件添加到源码库中，在 .gitignore 文件中加上 credentials.js。将凭证引入程序只需要这样做：

```
var credentials = require('./credentials.js');
```

我们后面还会用这个文件存放其他凭证，但现在只需要 cookie 秘钥。

如果你用的是示例项目配套的源码库，则必须自己创建一个 credentials.js 文件，因为这个文件不在源码库里。

9.2　Express中的Cookie

在程序中开始设置和访问 cookie 之前，需要先引入中间件 `cookie-parser`。首先 `npm install --save cookie-parser`，然后：

```
app.use(require('cookie-parser')(credentials.cookieSecret));
```

完成这个之后，你就可以在任何能访问到响应对象的地方设置 cookie 或签名 cookie：

```
res.cookie('monster', 'nom nom');
res.cookie('signed_monster', 'nom nom', { signed: true });
```

签名 cookie 的优先级高于未签名 cookie。如果你将签名 cookie 命名为 `signed_monster`，那就不能用这个名字再命名未签名 cookie（它返回时会变成 `undefined`）。

要获取客户端发送过来的 cookie 的值（如果有的话），只需访问请求对象的 `cookie` 或 `signedCookie` 属性：

```
var monster = req.cookies.monster;
var signedMonster = req.signedCookies.monster;
```

任何字符串都可以作为 cookie 的名称。比如，我们可以用 'signed monster' 代替 'signed_monster'，但这样我们必须用括号才能取到 cookie：req.signedCookies['signed monster']。因此我建议不要在 cookie 的名称中使用特殊字符。

要删除 cookie，请用 res.clearCookie：

```
res.clearCookie('monster');
```

设置 cookie 时可以使用如下这些选项：

- domain

  控制跟 cookie 关联的域名。这样你可以将 cookie 分配给特定的子域名。注意，你不能给 cookie 设置跟服务器所用域名不同的域名，因为那样它什么也不会做。

- path

  控制应用这个 cookie 的路径。注意，路径会隐含地通配其后的路径。如果你用的路径是 /（默认值），它会应用到网站的所有页面上。如果你用的路径是 /foo，它会应用到 /foo、/foo/bar 等路径上。

- maxAge

  指定客户端应该保存 cookie 多长时间，单位是毫秒。如果你省略了这一选项，浏览器关闭时 cookie 就会被删掉。（你也可以用 expiration 指定 cookie 过期的日期，但语法很麻烦。我建议用 maxAge。）

- secure

  指定该 cookie 只通过安全（HTTPS）连接发送。

- httpOnly

  将这个选项设为 true 表明这个 cookie 只能由服务器修改。也就是说客户端 JavaScript 不能修改它。这有助于防范 XSS 攻击。

- signed

  设为 true 会对这个 cookie 签名，这样就需要用 res.signedCookies 而不是 res.cookies 访问它。被篡改的签名 cookie 会被服务器拒绝，并且 cookie 值会重置为它的原始值。

9.3　检查Cookie

作为测试的一部分，你可能想要一种检查系统中 cookie 的方法。大多数浏览器都可以查看单个 cookie 和它们存储的值。在 Chrome 中，打开开发者工具，选择 Resources 标签，然后找到左侧树中的 Cookies 一项。展开它，你会看到当前访问的网站。点击它，你会看到

所有跟这个网站关联的 cookie。你也可以右键点击域名清除所有的 cookie，或者右键点击单个 cookie 移除它。

9.4 会话

会话实际上只是更方便的状态维护方法。要实现会话，必须在客户端存些东西，否则服务器无法从一个请求到下一个请求中识别客户端。通常的做法是用一个包含唯一标识的 cookie，然后服务器用这个标识获取相应的会话信息。cookie 不是实现这个目的的唯一手段，在“cookie 恐慌”的高峰时期（当时 cookie 滥用的情况非常猖獗），很多用户直接关掉了 cookie，因此发明了其他维护状态的方法，比如在 URL 中添加会话信息。这些技术混乱、困难且效率低下，所以最好别用。HTML5 为会话提供了另一种选择，那就是本地存储，但现在还没有令人叹服的理由去采用这种技术而放弃经过验证有效的 cookie。

从广义上来说，有两种实现会话的方法：把所有东西都存在 cookie 里，或者只在 cookie 里存一个唯一标识，其他东西都存在服务器上。前一种方式被称为“基于 cookie 的会话”，并且仅仅表示比使用 cookie 便利。然而，它还意味着要把你添加到 cookie 中的所有东西都存在客户端浏览器中，所以我不推荐用这种方式。只有在你知道自己只存少量信息，并且不介意用户能够访问这些信息，而它也不会随着时间的增长而失控时，你才可以用这种方式。如果你想采取这种方式，请查阅中间件 `cookie-session`（https://www.npmjs.org/package/cookie-session）。

9.4.1 内存存储

如果你更愿意把会话信息存在服务器上，这也是我推荐的方式，那么你必须找个地方存储它。入门级的选择是内存会话。它们非常容易设置，但也有个巨大的缺陷：重启服务器（你在本书中会做很多次）后会话信息就消失了。更糟的是，如果你扩展了多台服务器（参见第 12 章），那么每次请求可能是由不同的服务器处理的，所以会话数据有时在那里，有时不在。这明显是不可接受的用户体验。然而出于开发和测试的需要，有它就足够了。我们会在第 13 章介绍如何永久地存储会话信息。

首先安装 `express-session`（`npm install --save express-session`）。然后，在链入 `cookie-parser` 之后链入 `express-session`：

```
app.use(require('cookie-parser')(credentials.cookieSecret));
app.use(require('express-session')());
```

中间件 `express-session` 接受带有如下选项的配置对象：

- `key`

  存放唯一会话标识的 cookie 名称。默认为 `connect.sid`。

- `store`

  会话存储的实例。默认为一个 `MemoryStore` 的实例，可以满足我们当前的要求。第 13 章将会介绍如何使用数据库存储。

- `cookie`

  会话 cookie 的 cookie 设置（`path`、`domain`、`secure` 等）。适用于常规的 cookie 默认值。

9.4.2 使用会话

会话设置好以后，使用起来就再简单不过了，只是使用请求对象的 `session` 变量的属性：

```
req.session.userName = 'Anonymous';
var colorScheme = req.session.colorScheme || 'dark';
```

注意，对于会话而言，我们不是用请求对象获取值，用响应对象设置值，它全都是在请求对象上操作的。（响应对象没有 `session` 属性。）要删除会话，可以用 JavaScript 的 `delete` 操作符：

```
req.session.userName = null;            // 这会将 'userName' 设为 null
                                        // 但不会移除它
delete req.session.colorScheme;         // 这会移除 'colorScheme'
```

9.5 用会话实现即显消息

“即显”消息（不要跟 Adobe Flash 搞混了）只是在不破坏用户导航的前提下向用户提供反馈的一种办法。用会话实现即显消息是最简单的方式（也可以用查询字符串，但那样除了 URL 会更丑外，还会把即显消息放到书签里，这也许不是你想要的结果）。我们先把 HTML 设置好。我们将会用 Bootstrap 的警告消息组件显示我们的即显消息，所以请确保你引入了 Bootstrap。在你的模板文件里，找个醒目的地方（一般是直接放在网站的标题下面），添加下面的代码：

```
{{#if flash}}
        <div class="alert alert-dismissible alert-{{flash.type}}">
                <button type="button" class="close"
                        data-dismiss="alert" aria-hidden="true">&times;<button>
                <strong>{{flash.intro}}</strong> {{{flash.message}}}
        </div>
{{/if}}
```

注意，我们在 `flash.message` 外面用了 3 个大括号，这样我们就可以在消息中使用简单的 HTML（可能是要加重单词或包含超链接）。接下来添加一些中间件，如果会话中有 `flash` 对象，将它添加到上下文中。即显消息显示过一次之后，我们就要从会话中去掉它，以免它在下一次请求时再次显示。在路由之前添加下面这段代码：

```
app.use(function(req, res, next){
        // 如果有即显消息，把它传到上下文中，然后清除它
        res.locals.flash = req.session.flash;
        delete req.session.flash;
        next();
});
```

接下来我们看一下如何使用即显消息。假设我们的用户订阅了简报，并且我们想在用户订阅后把他们重定向到简报归档页面去。我们的表单处理器可能是这样的：

```
app.post('/newsletter', function(req, res){
    var name = req.body.name || '', email = req.body.email || '';
    // 输入验证
    if(!email.match(VALID_EMAIL_REGEX)) {
        if(req.xhr) return res.json({ error: 'Invalid name email address.' });
        req.session.flash = {
            type: 'danger',
            intro: 'Validation error!',
            message: 'The email address you entered was not valid.',
        };
        return res.redirect(303, '/newsletter/archive');
    }
    new NewsletterSignup({ name: name, email: email }).save(function(err){
        if(err) {
            if(req.xhr) return res.json({ error: 'Database error.' });
            req.session.flash = {
                type: 'danger',
                intro: 'Database error!',
                message: 'There was a database error; please try again later.',
            }
            return res.redirect(303, '/newsletter/archive');
        }
        if(req.xhr) return res.json({ success: true });
        req.session.flash = {
            type: 'success',
            intro: 'Thank you!',
            message: 'You have now been signed up for the newsletter.',
        };
        return res.redirect(303, '/newsletter/archive');
    });
});
```

注意看如何用同一个处理器处理 AJAX 提交（因为我们检查了 `req.xhr`），并且我们仔细地区分开了输入验证错误和数据库错误。记住，即便我们在前端做了输入验证（你应该这样做），在后台也应该再做一次，因为恶意用户能够绕过前端验证。

即显消息是网站中一种很棒的机制，即便在某些特定区域其他方法更合适一些（比如，即显消息在多表单“向导”或购物车结账流程中就不太合适）。即显消息在开发过程中也表现得很好，因为它们是一种简易的反馈方式，即便你之后会用其他技术取代它们。在搭建网站时，我首先要做的事情之一就是添加对即显消息的支持，并且本书后续会一直使用这一技术。

因为在中间件里把即显消息从会话中传给了 `res.locals.flash`，所以必须执行重定向以便显示即显消息。如果你不想通过重定向显示即显消息，直接设定 `res.locals.flash`，而不是 `req.session.flash`。

9.6　会话的用途

当你想跨页保存用户的偏好时，可以用会话。会话最常见的用法是提供用户验证信息，你登录后就会创建一个会话。之后你就不用在每次重新加载页面时再登录一次。即便没有用户账号，会话也有用。网站一般都要记住你喜欢如何排列东西，或者你喜欢哪种日期格式，这些都不需要登录。

尽管我建议你优先选择会话而不是 cookie，但理解 cookie 的工作机制也很重要（特别是因为有 cookie 才能用会话）。它对于你在应用中诊断问题、理解安全性及隐私问题都有帮助。

第 10 章

中间件

现在我们对中间件已经有了一些了解，我们使用过已有的中间件（例如，`body-parser`、`cookie-parser`、`static` 和 `connect-session`），甚至还自己写了一些（当我们检查查询字符串中有没有 `&test=1` 时，还有我们的 404 处理器）。但中间件究竟是什么？

从概念上讲，中间件是一种功能的封装方式，具体来说就是封装在程序中处理 HTTP 请求的功能。从实战上讲，中间件只是一个有 3 个参数的函数：一个请求对象、一个响应对象和一个 `next` 函数，稍后会作解释。（还有一种 4 个参数的形式，用来做错误处理，这会在本章末尾讲到。）

中间件是在管道中执行的。你可以想象一个送水的真实管道。水从一端泵入，然后在到达目的地之前还会经过各种仪表和阀门。这个比喻中很重要的一部分是顺序问题，你把压力表放在阀门之前和之后的效果是不同的。同样，如果你有个向水中注入什么东西的阀门，这个阀门“下游”的所有东西都会含有这个新添加的原料。在 Express 程序中，通过调用 `app.use` 向管道中插入中间件。

在 Express 4.0 之前，这个管道有些复杂，因为必须显式地把路由器连进来。取决于你在哪里连入路由器，路由的连入可以不按顺序来，这使得当你把中间件和路由处理器混在一起时，管道的顺序就更不清晰了。在 Express 4.0 中，中间件和路由处理器是按它们的连入顺序调用的，顺序更清晰。

在管道的最后放一个“捕获一切”请求的处理器是常见的做法，由它来处理跟前面其他所有路由都不匹配的请求。这个中间件一般会返回状态码 404（未找到）。

那么请求在管道中如何“终止”呢？这是由传给每个中间件的 `next` 函数来实现的。如果不

调用 next()，请求就在那个中间件中终止了。

学习如何灵活地考虑中间件和路由处理器是了解 Express 如何工作的关键。你应该把下面这些重点铭记于心。

- 路由处理器（app.get、app.post 等，经常被统称为 app.VERB）可以被看作只处理特定 HTTP 谓词（GET、POST 等）的中间件。同样，也可以将中间件看作可以处理全部 HTTP 谓词的路由处理器（基本上等同于 app.all，可以处理任何 HTTP 谓词；对于 PURGE 之类特别的谓词会有细微的差别，但对于普通的谓词而言，效果是一样的）。
- 路由处理器的第一个参数必须是路径。如果你想让某个路由匹配所有路径，只需用 /*。中间件也可以将路径作为第一个参数，但它是可选的（如果忽略这个参数，它会匹配所有路径，就像指定了 /* 一样）。
- 路由处理器和中间件的参数中都有回调函数，这个函数有 2 个、3 个或 4 个参数（从技术上讲也可以有 0 或 1 个参数，但这些形式没有意义）。如果有 2 个或 3 个参数，头两个参数是请求和响应对象，第三个参数是 next 函数。如果有 4 个参数，它就变成了错误处理中间件，第一个参数变成了错误对象，然后依次是请求、响应和 next 对象。
- 如果不调用 next()，管道就会被终止，也不会再有处理器或中间件做后续处理。如果你不调用 next()，则应该发送一个响应到客户端（res.send、res.json、res.render 等）；如果你不这样做，客户端会被挂起并最终导致超时。
- 如果调用了 next()，一般不宜再发送响应到客户端。如果你发送了，管道中后续的中间件或路由处理器还会执行，但它们发送的任何响应都会被忽略。

如果你想实际看一下，我们来尝试一些非常简单的中间件：

```
app.use(function(req, res, next){
        console.log('processing request for "' + req.url + '"....');
        next();
});

app.use(function(req, res, next){
        console.log('terminating request');
        res.send('thanks for playing!');
        // 注意，我们没有调用 next()……这样请求处理就终止了
});

app.use(function(req, res, next){
        console.log('whoops, i\'ll never get called!');
});
```

这里有三个中间件。第一个只是在将请求传给下一个中间件之前记录一条消息。然后下一个中间件会真正地处理请求。注意，如果我们忽略了 res.send，则不会有响应返回到客户端，最终会导致客户端超时。最后一个中间件永远也不会执行，因为所有请求都在前一个中间件中终止了。

接下来我们看一个更复杂、更完整的例子：

```
var app = require('express')();

app.use(function(req, res, next){
        console.log('\n\nALLWAYS');
        next();
});

app.get('/a', function(req, res){
        console.log('/a: 路由终止 ');
        res.send('a');
});
app.get('/a', function(req, res){
        console.log('/a: 永远不会调用 ');
});
app.get('/b', function(req, res, next){
        console.log('/b: 路由未终止 ');
        next();
});
app.use(function(req, res, next){
        console.log('SOMETIMES');
        next();
});
app.get('/b', function(req, res, next){
        console.log('/b (part 2): 抛出错误 ' );
        throw new Error('b 失败 ');
});

app.use('/b', function(err, req, res, next){
        console.log('/b 检测到错误并传递 ');
        next(err);
});
app.get('/c', function(err, req){
        console.log('/c: 抛出错误 ');
        throw new Error('c 失败 ');
});
app.use('/c', function(err, req, res, next){
        console.log('/c: 检测到错误但不传递 ');
        next();
});

app.use(function(err, req, res, next){
        console.log(' 检测到未处理的错误 : ' + err.message);
        res.send('500 - 服务器错误 ');
});

app.use(function(req, res){
        console.log(' 未处理的路由 ');
        res.send('404 - 未找到 ');
});

app.listen(3000, function(){
        console.log(' 监听端口 3000');
});
```

在尝试这个例子之前，先试试看能否猜出结果。路由有什么不同？客户端会看到什么？控

制台会输出什么？如果你能正确回答这些问题，就说明你已经掌握 Express 中的路由了。要特别注意请求 /b 和请求 /c 的差异，在这两个实例中都有一个错误，但一个结果是 404，另一个是 500。

注意，中间件必须是一个函数。记住，在 JavaScript 中，从一个函数中返回一个函数十分容易（并且常见）。例如，你会注意到 express.static 是一个函数，但我们真的会调用它，所以它必须返回另一个函数。看一下：

```
app.use(express.static);         // 这个不会像我们期望的那样工作
console.log(express.static());   // 将会输出 "function"，表明
                                 // express.static 是一个会返回函数的函数
```

还要注意，模块可以输出一个函数，而这个函数又可以直接用作中间件。例如，这里有个 lib/tourRequiresWaiver.js 模块（草地鹨旅行社的攀岩包需要一个责任免除条款）：

```
module.exports = function(req,res,next){
	var cart = req.session.cart;
	if(!cart) return next();
	if(cart.some(function(item){ return item.product.requiresWaiver; })){
		if(!cart.warnings) cart.warnings = [];
		cart.warnings.push('One or more of your selected tours' +
			'requires a waiver.');
	}
	next();
}
```

我们可以这样引入这个中间件：

```
app.use(require('./lib/requiresWaiver.js'));
```

不过更常见的做法是输出一个以中间件为属性的对象。例如，我们把所有购物车验证代码放在 lib/cartValidation.js 中：

```
module.exports = {
	checkWaivers: function(req, res, next){
		var cart = req.session.cart;
		if(!cart) return next();
		if(cart.some(function(i){ return i.product.requiresWaiver; })){
			if(!cart.warnings) cart.warnings = [];
			cart.warnings.push('One or more of your selected ' +
				'tours requires a waiver.');
		}
		next();
	},

	checkGuestCounts: function(req, res, next){
		var cart = req.session.cart;
		if(!cart) return next();
		if(cart.some(function(item){ return item.guests >
				item.product.maximumGuests; })){
```

```
                    if(!cart.errors) cart.errors = [];
                    cart.errors.push('One or more of your selected tours ' +
                        'cannot accommodate the number of guests you ' +
                        'have selected.');
                }
                next();
            }
        }
```

然后可以像以下这样连入中间件：

```
var cartValidation = require('./lib/cartValidation.js');

app.use(cartValidation.checkWaivers);
app.use(cartValidation.checkGuestCounts);
```

> 在前面的例子中，我们的中间件会用语句 `return next()` 提前终止。Express 不期望中间件返回值（并且它不会用返回值做任何事情），所以这只是缩短了的 `next(); return;`。

10.1 常用中间件

在 Express 4.0 之前，Express 中捆绑了 Connect，它包含了大部分常用的中间件。因为 Express 的捆绑方式，看起来这些中间件就像是 Express 的一部分一样（比如你可以这样引入 body-parser：`app.use(express.bodyParser)`）。这样看不出来这个中间件实际上是 Connect 的一部分。到 Express 4.0，Connect 从 Express 中移除了。随着这个改变，一些 Connect 中间件（比如 `body-parser`）也从 Connect 中分离出来变成了独立的项目。唯一保留在 Express 中的中间件只剩下 `static` 了。从 Express 中剥离中间件可以让 Express 不用再维护那么多的依赖项，并且这些独立的项目可以独立于 Express 而自行发展成熟。

大多数之前捆绑在 Express 中的中间件都十分基础，所以一定要知道“它去哪了”以及如何得到它。你大概总是需要 Connect，所以我建议你把它和 Express 一起安装（`npm install --save connect`），并使它在你的程序中可以访问到（`var connect = require(connect);`）。

- `basicAuth`（`app.use(connect.basicAuth)();`）
  提供基本的访问授权。记住，basic-auth 只提供最基本的安全，并且你只能通过 HTTPS 使用 basic-auth（否则用户名和密码是通过明文传输的）。只有在需要又快又容易的东西，并且在使用 HTTPS 时，才应该用 basic-auth。

- `body-parser`（`npm install --save body-parser, app.use(require(body- parser)());`）
  只连入 `json` 和 `urlencoded` 的便利中间件。这个中间件还在 Connect 里，但到 3.0 时会移除出去，所以建议你现在开始用这个包。除非你有特别的理由要分别单独使用 `json` 或 `urlencoded`，否则最好用这个包。

- json（参见body-parser）

  解析 JSON 编码的请求体。如果你在编写一个期望收到 JSON 编码请求体的 API，就会需要这个中间件。目前它的使用还不是十分普遍（大多数 API 仍然使用 application/x-www-form-urlencoded，这种编码可以被 urlencoded 中间件解析），但它确实能让你的程序更健壮，并不会过时。

- urlencoded（参见body-parser）

  解析互联网媒体类型为 application/x-www-form-urlencoded 的请求体。这是处理表单和 AJAX 请求最常用的方式。

- multipart（已废弃）

  解析互联网媒体类型为 multipart/form-data 的请求体。这个中间件已被废弃了，并在 Connect 3.0 中会被移除。你应该用 Busboy 或 Formidable 代替它（见第 8 章）。

- compress（app.use(connect.compress);）

  用 gzip 压缩响应数据。这是好事，用户会因此感激你的，特别是那些网络比较慢或者用手机上网的用户。它应该在任何可能会发送响应的中间件之前被尽早连入。唯一应该出现在 compress 之前的中间件只有 debugging 或 logging（它们不发送响应）。

- cookie-parser（npm install --save cookie-parser, app.use(require(cookie-parser)（秘钥放在这里）;)

  提供对 cookie 的支持。参见第 9 章。

- cookie-session（npm install --save cookie-session, app.use(require(cookie-session)()); ）

  提供 cookie 存储的会话支持。我一般不推荐使用这种存储方式的会话。你一定要把它放在 cookie-parser 后面连入。参见第 9 章。

- express-session（npm install --save express-session, app.use(require(express-session)());）

  提供会话 ID（存在 cookie 里）的会话支持。默认存在内存里，不适用于生产环境，并且可以配置为使用数据库存储。参见第 9 章和第 13 章。

- csurf（npm install --save csurf, app.use(require(csurf)());）

  防范跨域请求伪造（CSRF）攻击。因为它要使用会话，所以必须放在 express-session 中间件后面。它目前等同于 connect.csrf 中间件。可惜简单连入这个中间件并不能神奇地防范 CSRF 攻击，详情请参见第 18 章。

- directory（app.use(connect.directory());）

  提供静态文件的目录清单支持。如果不需要目录清单，则无需引入这个中间件。

- `errorhandler`（`npm install --save errorhandler, app.use(require(errorhandler)());`）
  为客户端提供栈追踪和错误消息。我建议不要在生产环境中连入它，因为它会暴露实现细节，可能引发安全或隐私问题。详情请参见第 20 章。

- `static-favicon`（`npm install --save static-favicon, app.use(require(static-favicon)(path_to_favicon));`）
  提供 favicon（出现在浏览器标题栏上的图标）。这个中间件不是必需的，你可以简单地在 static 目录下放一个 favicon.ico，但这个中间件能提升性能。如果你要使用它，应该尽可能地往中间件栈的上面放。你也可以使用除 favicon.ico 之外的其他文件名。

- `morgan`（之前的`logger`，`npm install --save morgan, app.use(require(morgan)());`）
  提供自动日志记录支持：所有请求都会被记录。详情请参见第 20 章。

- `method-override`（`npm install --save method-override, app.use(require(method-override)());`）
  提供对 `x-http-method-override` 请求头的支持，允许浏览器“假装”使用除 `GET` 和 `POST` 之外的 HTTP 方法。这对调试有帮助。只在编写 API 时才需要。

- `query`
  解析查询字符串，并将其变成请求对象上的 `query` 属性。这个中间件是由 Express 隐含连入的，所以不要自己连入它。

- `response-time`（`npm install --save response-time, app.use(require(response-time)());`）
  向响应中添加 `X-Response-Time` 头，提供以毫秒为单位的响应时间。一般在做性能调优时才需要这个中间件。

- `static`（`app.use(express.static(path_to_static_files)());`）
  提供对静态（public）文件的支持。这个中间件可以连入多次，并可指定不同的目录。详情请参见第 16 章。

- `vhost`（`npm install --save vhost, var vhost = require(vhost);`）
  虚拟主机（vhost），这个术语是从 Apache 借来的，它可使子域名在 Express 中更容易管理。详情请参见第 14 章。

10.2 第三方中间件

目前还没有第三方中间件的“商店”或索引目录。然而，几乎所有的 Express 中间件都能在 npm 上找到，所以如果你用 npm 搜索“Express”“Connect”和“Middleware”，会得到一个相当不错的清单。

第 11 章

发送邮件

邮件是网站跟世界沟通的主要方式之一。从用户注册到密码重置，从促销邮件到问题通知，很多地方都需要发送邮件，因此发送邮件是个非常重要的功能。

Node 和 Express 都没有内置的邮件发送功能，所以必须使用第三方模块。我推荐 Andris Reinman 的 Nodemailer（https://npmjs.org/package/nodemailer）。在深入到 Nodemailer 的配置之前，我们先学习一些与邮件有关的基础知识。

11.1　SMTP、MSA和MTA

发送邮件的通用语言是简单邮件传输协议（SMTP）。尽管用 SMTP 直接发送一封邮件给接收者的邮件服务器是有可能的，但这通常是个非常糟糕的主意。除非你是像 Google 或 Yahoo! 那样的“值得信任的发送者”，否则邮件很可能会直接被扔进垃圾箱。用邮件提交代理（MSA）比较好，它会通过可信的渠道投递邮件，降低邮件被标记为垃圾邮件的可能性。除了确保邮件成功送达，MSA 还处理诸如临时故障造成的滋扰和退回的邮件。最后一个是邮件传输代理（MTA），它提供将邮件真正送到其最终目的地的服务。对于本书而言，MSA、MTA 和“SMTP 服务器”本质上是一样的。

所以你需要一个 MSA。最容易的入手方式是用免费的邮件服务，比如 Gmail、Hotmail、iCloud、SendGrid 或 Yahoo!。这是一个临时的解决方案，除了有限制（比如，Gmail 在 24 小时内只允许发送 500 封邮件，并且每封邮件的收件人不能超过 100 个），它还会暴露你的个人邮件地址。尽管你可以指定如何显示发件人，比如 joe@meadowlarktravel.com，但粗略地看一下邮件头信息就能看出它是由 joe@gmail.com 发送的，非常不专业。一旦你

准备好进入生产环境，可以切换到 Sendgrid 或亚马逊简单 Email 服务（SES）之类的专业 MSA。

如果你在一个组织中工作，组织本身可能有 MSA，你可以联系 IT 部门问问他们有没有 SMTP 中继发送自动化的邮件。

11.2　接收邮件

大部分网站只需要发送邮件，比如密码重置和促销邮件。然而有些程序也需要接收邮件。比如问题追踪系统，在有人更新问题时会发出一封邮件，如果你答复了那封邮件，这个问题会根据你的响应自动更新。

可惜接收邮件牵涉的内容更多，本书就不再展开讨论了。如果你需要这个功能，应该看看 Andris Reinman 的 SimpleSMTP（https://github.com/andris9/simplesmtp）或 Haraka（http://haraka.github.io/）。

11.3　邮件头

邮件消息由两部分组成：头部和主体（跟 HTTP 请求非常像）。头部包含与邮件有关的信息：谁发的、发给谁、接收日期、主题等。这些头信息一般由邮件程序显示给用户，但头信息不止这些。大多数 Email 客户端允许你查看头部。如果你从来没看过，我建议你看一下。头信息给了所有关于邮件如何到达你这里的信息，邮件经过的所有服务器和 MTA 都会在头部里列出来。

有些头信息经常令人吃惊，比如“from”地址，它可以由发送方任意设定。当你指定的“from”地址不是你发送邮件的账号时，经常被当作“欺诈”。没有什么会阻止你将邮件的“from”地址设为 Bill Gates billg@microsoft.com。我不是在建议你尝试这种行为，只是指出你可以完全按自己的想法设定特定的头信息。有时出于正当理由可以这样做，但你绝不可以滥用它。

然而你发送的邮件必须有“from”地址。有时这在发送自动邮件时会出现问题，因此经常会出现像“不要回复 do-not-reply@meadowlarktravel.com”之类的返回地址。不管你是想采取这种方式，还是将发送地址设为“草地鹨旅行社 info@meadowlarktravel.com”，都完全取决于你。不过如果你采用了后一种方式，就要准备好答复发给 info@meadowlarktravel.com 的邮件。

11.4　邮件格式

互联网刚出现的时候，所有邮件都是简单的 ASCII 文本。然而现在的世界已经发生了很大

的变化，人们想用不同的语言发送邮件，并且想做一些疯狂的事情，比如包含格式化的文本、图片和附件。事情从此开始变得一发不可收拾，邮件格式和编码是一种混乱的技术和标准。幸好我们不用真的去应对这些错综复杂的事物，Nodemailer 会帮我们处理好。

重要的是，你要知道邮件既可以是普通文本（Unicode），也可以是 HTML。

几乎所有现代的邮件程序都支持 HTML 邮件，所以用 HTML 作为邮件格式一般相当安全。然而仍然有“纯粹的文本主义者”会逃避 HTML 邮件，所以我建议总是包含文本和 HTML 两种格式的邮件。如果你不想同时写文本和 HTML 邮件，Nodemailer 支持一种快捷方式，它可以自动从 HTML 中生成普通文本版本的邮件。

11.5　HTML邮件

HTML 邮件这个主题可以写一本书。它不像给网站写 HTML 那么简单，大多数邮件客户端只支持一小部分 HTML。大多数情况下，你不得不像在 1996 年似的写 HTML，这太无趣了。特别是你必须用表格控制布局（此处应该有悲伤的配乐）。

如果你经历过 HTML 跟浏览器的兼容性问题，就会了解它多么让人头疼。邮件的兼容性问题更严重。幸好有东西可以帮助我们。

首先向你推荐 MailChimp 关于如何编写 HTML 邮件的优秀文章（http://kb.mailchimp.com/campaigns/ways-to-build/how-to-code-html-emails）。它很好地囊括了编写 HTML 邮件的基础知识，并解释了在写 HTML 邮件时应该记住的事情。

其次是 HTML Email Boilerplate（http://htmlemailboilerplate.com/），它真的能节省很多时间。它本质上是一个编写得非常良好并经过严格测试的 HTML 邮件模板。

最后，还有测试……如果你已经阅读完怎么撰写 HTML 邮件，并且正在用 HTML Email Boilerplate，测试是确保你的邮件不会搞坏 Lotus Notes 7（是的，还有人用它）的唯一办法。感觉就像装 30 种不同的邮件客户端来测试一个邮件？我可不想这样。好在有个很好的服务可以帮你做这件事：Litmus（https://litmus.com/email-testing）。这个服务不算贵，起价 80 元一个月。但如果你要发很多促销邮件，它很难胜任。

换句话说，如果你的格式很普通，则没必要使用 Litmus 这种昂贵的测试服务。如果你能坚持只使用头、粗体 / 斜体文本、水平分割线、图片链接之类的东西，那是相当安全的。

11.6　Nodemailer

首先要安装 Nodemailer 包：

```
npm install --save nodemailer
```

然后，引入 `nodemailer` 包并创建一个 Nodemailer 实例（按 Nodemailer 的说法是一个“传输”）：

```
var nodemailer = require('nodemailer');

var mailTransport = nodemailer.createTransport('SMTP',{
        service: 'Gmail',
        auth: {
                user: credentials.gmail.user,
                pass: credentials.gmail.password,
        }
});
```

注意，我们用到了第 9 章中设置的 credentials 模块。你需要对你的 credentials.js 做出相应的修改：

```
module.exports = {
        cookieSecret: 'your cookie secret goes here',
        gmail: {
                user: 'your gmail username',
                password: 'your gmail password',
        }
};
```

Nodemailer 为大多数流行的邮件服务提供了快捷方式：Gmail、Hotmail、iCloud、Yahoo!，除此之外还有很多。如果你的 MSA 没有出现在这个列表上，或者你需要直接连接一个 SMTP 服务器，它也支持：

```
var mailTransport = nodemailer.createTransport('SMTP',{
        host: 'smtp.meadowlarktravel.com',
        secureConnection: true,     // 用 SSL 端口：465
        auth: {
                user: credentials.meadowlarkSmtp.user,
                pass: credentials.meadowlarkSmtp.password,
        }
});
```

11.6.1 发送邮件

现在有了邮件传输实例，我们可以发送邮件了。我们会从一个非常简单的例子开始，向一个接收者发送文本邮件：

```
mailTransport.sendMail({
        from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
        to: 'joecustomer@gmail.com',
        subject: 'Your Meadowlark Travel Tour',
        text: 'Thank you for booking your trip with Meadowlark Travel.'+
                'We look forward to your visit!',
}, function(err){
        if(err) console.error( 'Unable to send email: ' + error );
});
```

你会注意到我们在这里处理了错误，但你也应该知道，没有错误不一定表示邮件成功发给了接收者：只有在跟 MSA 通信出现问题时才会设置回调函数的 err 参数（比如网络或授权错误）。如果 MSA 不能投递邮件（比如因为无效的邮件地址或者未知的用户），你会收到一封投递给 MSA 账号的失败邮件（比如你用自己的个人 Gmail 作为 MSA，你的 Gmail 收件箱中就会收到一封失败消息）。

如果你需要系统自动判断邮件是否投递成功，有两个选择。一是使用支持错误报告的 MSA。亚马逊的简单邮件服务（SES）就是这样的服务，并且邮件退信通知是通过他们的简单通知服务（SNS）发送的，你可以配置其调用运行在你网站上的 Web 服务。另一个选择是使用直接投递，跳过 MSA。我不推荐使用直接投递，因为它是一个复杂的方案，并且你的邮件很可能会被标记为垃圾邮件。这些选择都不简单，并且都超出了本书的范围。

11.6.2 将邮件发送给多个接收者

Nodemail 支持发送邮件给多个接收者，只要把他们用逗号分开：

```
mailTransport.sendMail({
        from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
        to: 'joe@gmail.com, "Jane Customer" <jane@yahoo.com>, ' +
                'fred@hotmail.com',
        subject: 'Your Meadowlark Travel Tour',
        text: 'Thank you for booking your trip with Meadowlark Travel.  ' +
                'We look forward to your visit!',
}, function(err){
        if(err) console.error( 'Unable to send email: ' + error );
});
```

注意，在这个例子中，我们把普通邮件地址（joe@gmail.com）和指定了接收者姓名的地址（“Jane Customer” jane@yahoo.com）混在了一起。这种语法是可以的。

在向多个接收者发送邮件时，你必须注意观察 MSA 的限制。比如 Gmail，每封邮件的接收者上限是 100 个。即便更强壮的服务，比如 SendGrid，也会限制接收者的数量（SendGrid 建议每封邮件的接收者不超过 1000 个）。如果你发送批量邮件，可能要发送多条消息，每条消息有多个接收者：

```
// largeRecipientList 是一个邮件地址数组
var recipientLimit = 100;
for(var i=0; i<largeRecipientList.length/recipientLimit; i++){
        mailTransport.sendMail({
                from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
                to: largeRecipientList
                        .slice(i*recipientLimit, i*(recipientLimit+1)).join(','),
                subject: 'Special price on Hood River travel package!',
                text: 'Book your trip to scenic Hood River now!',
        }, function(err){
                if(err) console.error( 'Unable to send email: ' + error );
        });
}
```

11.7 发送批量邮件的更佳选择

尽管你确实可以通过 Nodemailer 和恰当的 MSA 发送批量邮件，但在这样做之前你应该细心考虑。一个负责任的邮件营销必须提供一种退订营销邮件的办法，并且这不是个轻而易举的任务。还要乘以你维护的每个订阅列表（比如，你可能有一个周简讯和一个特殊的公告营销）。这是一个最好不要白费力气做重复工作的领域。像 MailChimp（http://mailchimp.com/）和 Campaign Monitor（http://www.campaignmonitor.com/）之类的服务提供了你需要的一切，包括监测邮件营销成功情况的优秀工具。你完全负担得起，我强烈推荐使用它们做营销邮件、简讯等。

11.8 发送HTML邮件

我们已经发过普通文本的邮件了，但现在大多数人都想看到更漂亮的邮件。Nodemailer 允许你在同一封邮件里发送 HTML 和普通文本两种版本，让邮件客户端选择显示哪个版本（一般是 HTML）：

```
mailTransport.sendMail({
        from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
        to: 'joecustomer@gmail.com, "Jane Customer" ' +
                '<janecustomer@gyahoo.com>, frecsutomer@hotmail.com',
        subject: 'Your Meadowlark Travel Tour',
        html: '<h1>Meadowlark Travel</h1>\n<p>Thanks for book your trip with ' +
                'Meadowlark Travel.  <b>We look forward to your visit!</b>',
        text: 'Thank you for booking your trip with Meadowlark Travel.  ' +
                'We look forward to your visit!',
}, function(err){
        if(err) console.error( 'Unable to send email: ' + error );
});
```

这个工作量很大，所以我不推荐这种方式。幸好 Nodemailer 会自动将 HTML 翻译成普通文本，如果你要求它那么做：

```
mailTransport.sendMail({
        from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
        to: 'joecustomer@gmail.com, "Jane Customer" ' +
                '<janecustomer@gyahoo.com>, frecsutomer@hotmail.com',
        subject: 'Your Meadowlark Travel Tour',
        html: '<h1>Meadowlark Travel</h1>\n<p>Thanks for book your trip with ' +
                'Meadowlark Travel. <b>We look forward to your visit!</b>',
        generateTextFromHtml: true,
}, function(err){
        if(err) console.error( 'Unable to send email: ' + error );
});
```

11.8.1 HTML邮件中的图片

尽管可以在 HTML 邮件中嵌入图片，但我强烈反对这样做，因为它们会使你的邮件变得臃

肿，并且一般会被当成不好的做法。相反，你应该把用在邮件中的图片放在 Web 服务器上，并在邮件中放入正确的链接。

你最好在静态资源文件夹中给邮件图片一个专门的位置。你甚至应该把同时用在网站和邮件中的资源文件（比如你的日志）分开，这样会减小你的邮件布局受到负面影响的可能性。

我们给草地鹨旅行社项目添加一些邮件资源。在 public 目录下创建一个子目录 email。你可以把 logo.png 以及你想要放在邮件中的其他任何图片放在那里。然后你可以在邮件中直接使用那些图片：

```
<img src="//meadowlarktravel.com/email/logo.png" alt="Meadowlark Travel">
```

> 当你发送邮件给其他人时，很明显不应该用 localhost，他们甚至可能不会有服务器在运行，更别说是运行在端口 3000 上了。根据你所用的邮件客户端，或许可以在你的邮件中用 localhost 来进行测试，但在你的机器之外是行不通的。第 16 章我们会探讨一些平滑地从开发转向生产环境的技术。

11.8.2 用视图发送HTML邮件

之前我们把 HTML 字符串放到了 JavaScript 中，你应该尽量避免这种做法。现在我们的 HTML 还很简单，但看看 HTML Email Boilerplate（http://htmlemailboilerplate.com/），你想把那些套路化代码都放到字符串里吗？绝不可能。

好在我们可以用视图处理这个问题。我们考虑一下“感谢您预订草地鹨旅行社的旅游产品”这个邮件的例子，稍微扩展一点。假设我们有一个购物车对象，它包含了我们的订单信息。这个购物车对象会存在于会话中。订单流程中的最后一步是由 /cart/chckout 处理的表单，它会发送一封确认邮件。我们先从创建“感谢”页面的视图开始，views/cart-thank-you.handlebars：

```
<p>Thank you for booking your trip with Meadowlark Travel, {{cart.billing.name}}!</p>
<p>Your reservation number is {{cart.number}}, and an email has been sent to
{{cart.billing.email}} for your records.</p>
```

然后创建一个邮件模板。下载 HTML Email Boilerplate，把它放到 iews/email/cart-thank-you.handlebars 中。编辑这个文件，修改主体部分：

```
<body>
<table cellpadding="0" cellspacing="0" border="0" id="backgroundTable">
    <tr>
        <td valign="top">
            <table cellpadding="0" cellspacing="0" border="0" align="center">
                <tr>
                    <td width="200" valign="top"><img class="image_fix"
```

```
                        src="http://meadowlarktravel.com/email/logo.png"
                        alt="Meadowlark Travel" title="Meadowlark Travel"
                        width="180" height="220" /></td>
                </tr>
                <tr>
                    <td width="200" valign="top"><p>
                        Thank you for booking your trip with Meadowlark Travel,
                        {{cart.billing.name}}.</p><p>Your reservation number
                        is {{cart.number}}.</p></td>
                </tr>
                <tr>
                    <td width="200" valign="top">Problems with your reservation?
                    Contact Meadowlark Travel at
                    <span class="mobile_link">555-555-0123</span>.</td>
                </tr>
            </table>
        </td>
    </tr>
</table>
</body>
```

> 因为你不能在邮件中用 localhost，所以如果你的网站还没建立起来，可以随便找个图片占位。比如 http://placehold.it/100x100 会为你动态提供一个 100 像素的方形图片。这项技术在只为占位（FPO）的图片和以布局为目的的情况下十分常用。

现在我们可以为购物车“感谢”页面创建路由：

```
app.post('/cart/checkout', function(req, res){
        var cart = req.session.cart;
        if(!cart) next(new Error('Cart does not exist.'));
        var name = req.body.name || '', email = req.body.email || '';
        // 输入验证
        if(!email.match(VALID_EMAIL_REGEX))
                return res.next(new Error('Invalid email address.'));
        // 分配一个随机的购物车 ID；一般我们会用一个数据库 ID
        cart.number = Math.random().toString().replace(/^0\.0*/, '');
        cart.billing = {
                name: name,
                email: email,
        };
    res.render('email/cart-thank-you',
        { layout: null, cart: cart }, function(err,html){
                if( err ) console.log('error in email template');
                mailTransport.sendMail({
                    from: '"Meadowlark Travel": info@meadowlarktravel.com',
                    to: cart.billing.email,
                    subject: 'Thank You for Book your Trip with Meadowlark',
                    html: html,
                    generateTextFromHtml: true
                }, function(err){
```

```
                    if(err) console.error('Unable to send confirmation: '
                        + err.stack);
                });
            }
        );
        res.render('cart-thank-you', { cart: cart });
    });
```

注意，我们调用了两次 res.render。一般只调用一次（调用两次只会显示第一次调用的结果）。然而在这个例子中，我们第一次调用避开了正常的渲染过程，注意我们提供了一个回调函数。这样可以防止视图的结果渲染到浏览器中。相反，回调函数在参数 html 中接收到渲染好的视图，我们只需要接受渲染好的 HTML 并发送邮件。我们指定了 layout: null 以防止使用我们的布局文件，因为它全在邮件模板中（另一种方式是为邮件单独创建一个模板）。最后我们再次调用了 res.render。这次结果会像往常一样将 HTML 响应发给浏览器。

11.8.3 封装邮件功能

如果你的网站上很多地方都要用邮件，你可能想把邮件的功能封装起来。假定你总想让网站从同一个发送者发送邮件（“草地鹨旅行社” info@meadowlarktravel.com），并且总想用自动生成的文本以 HTML 格式发送。创建模块 lib/email.js：

```
var nodemailer = require('nodemailer');

module.exports = function(credentials){

        var mailTransport = nodemailer.createTransport('SMTP',{
                service: 'Gmail',
                auth: {
                        user: credentials.gmail.user,
                        pass: credentials.gmail.password,
                }
        });

        var from = '"Meadowlark Travel" <info@meadowlarktravel.com>';
        var errorRecipient = 'youremail@gmail.com';

        return {
                send: function(to, subj, body){
                    mailTransport.sendMail({
                        from: from,
                        to: to,
                        subject: subj,
                        html: body,
                        generateTextFromHtml: true
                    }, function(err){
                        if(err) console.error('Unable to send email: ' + err);
                    });
```

```
                }),

                emailError: function(message, filename, exception){
                    var body = '<h1>Meadowlark Travel Site Error</h1>' +
                        'message:<br><pre>' + message + '</pre><br>';
                    if(exception) body += 'exception:<br><pre>' + exception
                        + '</pre><br>';
                    if(filename) body += 'filename:<br><pre>' + filename
                        + '</pre><br>';
                  mailTransport.sendMail({
                    from: from,
                    to: errorRecipient,
                    subject: 'Meadowlark Travel Site Error',
                    html: body,
                    generateTextFromHtml: true
                  }, function(err){
                    if(err) console.error('Unable to send email: ' + err);
                  });
                },
    }
```

现在要发送邮件，我们只需要：

```
var emailService = require('./lib/email.js')(credentials);

emailService.send('joecustomer@gmail.com', 'Hood River tours on sale today!',
        'Get \'em while they\'re hot!');
```

你会注意到我们还添加了一个 emailError 方法，这将在下一节讨论。

11.9　将邮件作为网站监测工具

如果网站出问题了，你是不是想赶在客户之前知道？或者赶在老板之前？一个好办法是让网站在出错时给你发消息。我们在前面那个例子中刚添加了这样一个方法，所以当网站中有错误时，你可以这样做：

```
if(err){
        email.sendError('the widget broke down!', __filename);
        // ……给用户显示错误消息
}

// 或者

try {
        // 在这里做些不确定的事情……
} catch(ex) {
        email.sendError('the widget broke down!', __filename, ex);
        // ……给用户显示错误消息
}
```

这不是日志的替代品，在第 12 章中我们会研究一个更强壮的日志和通知机制。

第12章

与生产相关的问题

你可能觉得现在开始讨论与生产相关的问题还为时尚早，但尽早考虑这些问题可以帮你节省很多时间，并减少你将会承受的痛苦，这些痛苦在正式启用那天总会不期而至。

本章会介绍 Express 对不同执行环境的支持、扩展网站的方法以及如何监控网站的的健康状况。我们将会看到如何模拟生产环境来进行测试和开发，以及如何执行压力测试，以便提前找出生产中的问题。

12.1　执行环境

Express 支持执行环境的概念，它是一种在生产、开发或测试模式中运行应用程序的方法。实际上你可以按自己的想法创建很多种不同的环境。比如，你可以有一个临时环境或训练环境。然而要记住，开发、生产和测试是“标准”环境，Express、Connect 以及第三方中间件可能会基于这些环境做出决定。换句话说，如果你有一个“临时”环境，则无法让它自动集成生产环境的属性。因此我建议你坚持使用标准的开发、生产和测试环境。

尽管可以调用 `app.set('env', 'production')` 指定执行环境，但我不建议你这样做，因为那意味着不管什么情况，你的应用程序都会一直运行在那个环境中。更糟的是，它可能在一个环境中开始运行，然后切换到另一个环境。

用环境变量 `NODE_ENV` 指定执行环境更好。我们来修改一下我们的应用程序，通过调用 `app.get('env')` 让它报告一下它运行在哪种模式下：

```
http.createServer(app).listen(app.get('port'), function(){
    console.log( 'Express started in ' + app.get('env') +
        ' mode on http://localhost:' + app.get('port') +
        '; press Ctrl-C to terminate.' );
});
```

如果你现在启动服务器，将会看到它运行在开发模式下，因为如果你没有指定，开发模式就是默认模式。我们试着把它放在生产模式下：

```
$ export NODE_ENV=production
$ node meadowlark.js
```

如果你用的是 Unix/BSD 系统或 Cygwin，这里有个方便的语法，让你仅为一次命令执行期间设定环境：

```
$ NODE_ENV=production node meadowlark.js
```

这会在生产模式下运行服务器，但当服务器终止时，环境变量 `NODE_ENV` 还是原来的值。

> 如果在生产模式下启动 Express，你可能会注意到有些组件不适合在生产模式下使用的警告信息。如果你一直在按照本书的例子做，会看到 `connect.session` 用了内存存储，这不适合生产环境。一旦我们到第 13 章切换到数据库存储，这个警告就会消失。

12.2 环境特定配置

只是改变执行环境起不到太大的作用，尽管 Express 在生产模式下会输出更多警告到控制台中（比如告诉你被废弃的模块将来会被移除）。还有，在生产模式下，视图缓存会默认启用（见第 7 章）。

执行环境大体是一个可以利用的工具，你可以轻松地决定应用程序在不同的环境下应该做何表现。给你一个忠告，尽量缩小开发、测试和生产环境之间的差异。也就是说你应该保守地使用这个功能。如果你的开发和测试环境跟生产环境差别很大，就会增加生产环境中表现不同的机会，这会导致更多的缺陷（或者很难找到）。有些差异是不可避免的，比如，如果你的程序是高度数据库驱动的，你可能不想在开发期间干扰生产数据库，并且这是环境特定配置的良好候选用途。另外一个影响不大的领域是更加详细的日志。你想在开发时记录的很多东西都没必要在生产环境中记录。

我们要给程序添加一些日志。在开发环境中，我们会用 Morgan（`npm install --save morgan`），它的输出是便于查看的彩色文本。在生产环境中，我们用 express-logger（`npm install --save express-logger`），它支持日志循环（每 24 小时复制一次，然后开始新的日志，防止日志文件无限制地增长）。接下来给程序文件添加日志支持：

```
switch(app.get('env')){
    case 'development':
        // 紧凑的、彩色的开发日志
        app.use(require('morgan')('dev'));
        break;
    case 'production':
        // 模块 'express-logger' 支持按日志循环
        app.use(require('express-logger')({
            path: __dirname + '/log/requests.log'
        }));
        break;
}
```

如果你要测试日志，可以在生产模式下运行程序（`NODE_ENV=production node meadowlark.js`）。如果你想实际看看日志的循环功能，可以编辑 node_modules/express-logger/logger.js，修改变量 `defaultInterval`，比如从 24 小时改成 10 秒（记住，修改 node_modules 中的包只能是出于实验或学习目的）。

> 在上面的例子中，我们用 `__dirname` 把请求日志存在项目自身的子目录下。如果采用这种方式，你应该把日志文件添加到 .gitignore 文件中。或者你可以采用 Unix 风格的方式，把日志文件放在 /var/log 的一个子目录下，像 Apache 默认做的那样。

我要再次强调一下，在做出与环境相关的配置选择时，你应该做出最佳判断。一定要记得，当网站启用时，你的生产实例会运行在生产模式下（或者应该是）。不管你什么时候试图做出与开发相关的修改，都应该先考虑可能会对生产环境产生的 QA 影响。我们会在第 13 章见到更加健壮的环境特定配置范例。

12.3 扩展你的网站

现在，扩展通常意味着向上扩展或向外扩展。向上扩展是指让服务器变得更强：更快的 CPU，更好的架构，更多内核，更多内存，等等。而向外扩展只是意味着更多的服务器。随着云计算的流行和虚拟化的普及，服务器和计算能力的相关性变得越来越小，并且对于网站的扩展需求而言，向外扩展是成本收益率更高的办法。

在用 Node 开发网站时，你应该总是考虑向外扩展的可能性。即便你的程序很小（甚至可能只是一个受众有限的内联网程序），并且你从来没想过需要扩展，考虑一下也是个好习惯。毕竟你的下一个 Node 项目可能是下一个 Twitter，向外扩展是必不可少的。好在 Node 对向外扩展支持得很好，并且带着这个想法写程序也不会觉得痛苦。

在搭建一个设计好要向外扩展的网站时，最重要的是持久化。如果你习惯于用基于文件的存储做持久化，那就此打住吧，因为那会让人发疯的。我第一次遇到这个问题几乎是场灾

难。我的一个客户运营着一个基于 Web 的竞赛，这个 Web 程序要通知前 50 名获胜者他们会收到奖励。对于那个客户来说，因为公司 IT 方面的某些限制，我们不能轻易使用数据库，所以大部分持久化是通过写入普通文件实现的。我像往常那样处理，把每条记录写到文件中。只要这个文件记录了 50 名获胜者，就不会再有人收到他们已经获胜的通知了。问题是服务器做了负载平衡，一半请求由一台服务器处理，另外一半由另外一台服务器处理。一台服务器通知 50 个人他们获胜了……另外一台也通知了。好在奖品不大（抓绒毛毯），不是 iPad，并且客户为此颁出了 100 份奖品（我提出来要为自己的错误负责，愿意承担另外 50 条毯子，但他们慷慨地拒绝了我的提议）。这个故事是要告诉你们，除非所有服务器都能访问到那个文件系统，否则你不应该用本地文件系统做持久化。不过只读数据是个例外，比如日志和备份。比如，我一般会把表单提交的数据备份到本地普通文件中，以防数据库接连失效。一旦遇到数据库中断的情况，到每个服务器上收集文件虽然麻烦，但最起码不会造成破坏。

12.3.1　用应用集群扩展

Node 本身支持应用集群，它是一种简单的、单服务器形式的向外扩展。使用应用集群，你可以为系统上的每个内核（CPU）创建一个独立的服务器（有更多的服务器而不是内核数不会提高程序的性能）。应用集群好在两个地方：第一，它有助于实现给定服务器性能的最大化（硬件或虚拟机）；第二，它是一种在并行条件下测试程序的低开销方式。

我们继续给网站添加集群支持。尽管在主程序文件中做这些工作的做法十分普遍，但我们准备创建第二个程序文件，用之前一直在用的非集群程序文件在集群中运行程序。为此我们必须先对 meadowlark.js 做些轻微的调整：

```
function startServer() {
    http.createServer(app).listen(app.get('port'), function(){
      console.log( 'Express started in ' + app.get('env') +
        ' mode on http://localhost:' + app.get('port') +
        '; press Ctrl-C to terminate.' );
    });
}

if(require.main === module){
    // 应用程序直接运行；启动应用服务器
    startServer();
} else {
    // 应用程序作为一个模块通过 "require" 引入 : 导出函数
    // 创建服务器
    module.exports = startServer;
}
```

这样修改之后，meadowlark.js 既可以直接运行（`node meadowlark.js`），也可以通过 `require` 语句作为一个模块引入。

当直接运行脚本时，require.main === module 是 true；如果它是 false，表明你的脚本是另外一个脚本用 require 加载进来的。

然后我们会创建一个新脚本，meadowlark_cluster.js：

```
var cluster = require('cluster');

function startWorker() {
    var worker = cluster.fork();
    console.log('CLUSTER: Worker %d started', worker.id);
}

if(cluster.isMaster){

    require('os').cpus().forEach(function(){
            startWorker();
    });

    // 记录所有断开的工作线程。如果工作线程断开了，它应该退出
    // 因此我们可以等待 exit 事件然后繁衍一个新工作线程来代替它
    cluster.on('disconnect', function(worker){
        console.log('CLUSTER: Worker %d disconnected from the cluster.',
            worker.id);
    });
    // 当有工作线程死掉（退出）时，创建一个工作线程代替它
    cluster.on('exit', function(worker, code, signal){
        console.log('CLUSTER: Worker %d died with exit code %d (%s)',
            worker.id, code, signal);
        startWorker();
    });

} else {

    // 在这个工作线程上启动我们的应用服务器，参见 meadowlark.js
    require('./meadowlark.js')();

}
```

在这个 JavaScript 执行时，它或者在主线程的上下文中（当用 node meadowlark_cluster.js 直接运行它时），或者在工作线程的上下文中（在 Node 集群系统执行它时）。属性 cluster.isMaster 和 cluster.isWorker 决定了你运行在哪个上下文中。在我们运行这个脚本时，它是在主线程模式下执行的，并且我们用 cluster.fork 为系统中的每个 CPU 启动了一个工作线程。我们还监听了工作线程的 exit 事件，重新繁衍死掉的工作线程。

最后，我们在 else 从句中处理工作线程的情况。既然我们将 meadowlark.js 配置为模块使用，只需要引入并立即调用它（记住，我们将它作为一个函数输出并启动服务器）。

现在启动新的集群化服务器：

```
node meadowlark_cluster.js
```

> 如果你用的是虚拟机（比如 Oracle 的 VirtualBox），则必须将 VM 配置为多个 CPU。虚拟机一般默认只有一个 CPU。

假定你在多核系统上，应该能看到一些工作线程启动了。如果你想看到不同工作线程处理不同请求的证据，在路由前添加下面这个中间件：

```
app.use(function(req,res,next){
    var cluster = require('cluster');
    if(cluster.isWorker) console.log('Worker %d received request',
        cluster.worker.id);
});
```

现在你可以用浏览器连接你的应用程序。刷新几次，看看你怎么能在每个请求上得到不同的工作线程。

12.3.2　处理未捕获的异常

在 Node 的异步世界中，未捕获的异常是特别需要关注的问题。我们先从一个不会引起太多麻烦的简单例子开始（我希望你能按照这些例子做）：

```
app.get('/fail', function(req, res){
    throw new Error('Nope!');
});
```

在 Express 执行路由处理器时，它把它们封装在一个 `try/catch` 块中，所以这不是一个真正的未捕获异常。这不会引起太多问题，Express 会在服务器端记录异常，并且访问者会得到一个丑陋的栈输出。然而服务器是稳定的，其他请求还能得到正确处理。如果我们想提供一个“好的”错误页面，可以创建文件 views/500.handlebars 并在所有路由后面添加一个错误处理器：

```
app.use(function(err, req, res, next){
    console.error(err.stack);
    app.status(500).render('500');
});
```

提供一个定制的错误页面总归是一个好的做法，当错误出现时，它不仅在用户面前显得更专业，还可以让你采取行动。比如，你可以在这个错误处理器中发送一封邮件给开发团队，让他们知道网站出错了。可惜这只能用在 Express 可以捕获的异常上。我们来尝试一些更糟的情况：

```
app.get('/epic-fail', function(req, res){
    process.nextTick(function(){
        throw new Error('Kaboom!');
    });
});
```

去试一下吧。结果相当糟糕，它把你的整个服务器都搞垮了。它不仅没向用户显示一个友好的错误信息，而且现在你的服务器还宕机了，不能再处理请求了。这是因为 setTimeout 是异步执行的，抛出异常的函数被推迟到 Node 空闲时才执行。问题是，当 Node 得到空闲可以执行这个函数时，它已经没有其所服务的请求的上下文了，所以它已经没有资源了，只能毫不客气地关掉整个服务器，因为现在它处于不确定的状态（Node 无法得知函数的目的，或者其调用者的目的，所以它不可能再假设后续函数还能正确工作）。

process.nextTick 跟调用没有参数的 setTimeout 非常像，但它效率更高。我们在这里用它是为了演示，一般你不会在服务器端代码里用它。然而在接下来的几章里，我们会处理很多异步执行的任务，比如数据库访问、文件系统访问和网络访问，并且它们都会遇到这个问题。

我们可以采取行动处理未捕获的异常，但如果 Node 不能确定程序的稳定性，你也不能。换句话说，如果出现了未捕获异常，唯一能做的也只是关闭服务器。在这种情况下，最好的做法就是尽可能正常地关闭服务器，并且有个故障转移机制。最容易的故障转移机制是使用集群（就像之前提到的）。如果你的程序是运行在集群模式下的，当一个工作线程死掉后，主线程会繁衍另一个工作线程来取代它。（你甚至不需要有多个工作线程，有一个工作线程的集群就够了，尽管那样故障转移可能会稍微有点慢。）

那么在遇到未处理异常时，我们怎么才能尽可能正常地关闭服务器呢？ Node 有两种机制解决这个问题：uncaughtException 事件和域。

使用域是较新的方式，也是推荐的方式（uncaughtException 甚至可能会在将来的 Node 版本中去掉）。一个域基本上是一个执行上下文，它会捕获在其中发生的错误。有了域，你在错误处理上可以更灵活，不再是只有一个全局的未捕获异常处理器，你可以有很多域，可以在处理易出错的代码时创建一个新域。

每个请求都在一个域中处理是一种好的做法，这样你就可以追踪那个请求中所有的未捕获错误并做出相应的响应（正常地关闭服务器）。添加一个中间件就可以非常轻松地满足这个要求。这个中间件应该在所有其他路由或中间件前面：

```
app.use(function(req, res, next){
    // 为这个请求创建一个域
    var domain = require('domain').create();
    // 处理这个域中的错误
    domain.on('error', function(err) {
```

```
            console.error('DOMAIN ERROR CAUGHT\n', err.stack);
            try {
                // 在 5 秒内进行故障保护关机
                setTimeout(function(){
                    console.error('Failsafe shutdown.');
                    process.exit(1);
                }, 5000);

                // 从集群中断开
                var worker = require('cluster').worker;
                if(worker) worker.disconnect();

                // 停止接收新请求
                server.close();

                try {
                    // 尝试使用 Express 错误路由
                    next(err);
                } catch(err) {
                    // 如果 Express 错误路由失效，尝试返回普通文本响应
                    console.error('Express error mechanism failed.\n', err.stack);
                    res.statusCode = 500;
                    res.setHeader('content-type', 'text/plain');
                    res.end('Server error.');
                }
            } catch(err){
                console.error('Unable to send 500 response.\n', err.stack);
            }
        });

        // 向域中添加请求和响应对象
        domain.add(req);
        domain.add(res);

        // 执行该域中剩余的请求链
        domain.run(next);
    });

    // 其他中间件和路由放在这里

    var server = http.createServer(app).listen(app.get('port'), function(){
        console.log('Listening on port %d.', app.get('port'));
    });
```

我们做的第一件事是创建一个域，然后在上面附着一个错误处理器。只要这个域中出现未捕获的错误，就会调用这个函数。我们在这里采取的方式是试图给任何处理中的请求以恰当的响应，然后关闭服务器。根据错误的性质，可能无法响应处理中的请求，所以我们首先要确立关闭服务器的截止时间。在这个例子中，我们允许服务器在 5 秒内响应处理中的请求（如果它可以）。你所选择的数值取决于你的程序，如果程序经常有长请求，你就应该给更多的时间。一旦确立了截止时间，我们会从集群中断开（如果在集群中），以防止集群给我们分配更多的请求。然后明确告诉服务器我们不再接受新的连接。最后，我们试

图传到错误处理路由（`next(err)`）来响应产生错误的请求。如果那会抛出错误，我们退回去用普通的 Node API 响应。如果其他的全部失败了，我们会记录错误（客户端得不到响应，最终会超时）。

一旦设置好未处理异常处理器，我们就把请求和响应对象添加到域中（允许那些对象上的所有方法抛出的错误都由域处理）。最后，我们在域的上下文中运行管道中的下一个中间件。注意，这可以有效地运行域中管道里的所有中间件，因为对 `next()` 的调用是链起来的。

如果搜索一下 npm，你会发现有些中间件提供了这个功能。然而了解域的错误处理机制非常重要，在有未捕获异常时关闭服务器也很重要。最后，"正常地关闭"的含义会随着你的部署配置而变化。比如，如果你限制只有一个工作线程，你可能想立即关闭，以正在进行中的所有会话为代价，然而如果你有多个工作线程，在关闭前就有了更多的回旋余地，让垂死的工作线程服务剩余的请求。

我强烈推荐你阅读 William Bert 的优秀文章"The 4 Keys to 100% Uptime with Node.js"（http://engineering.fluencia.com/blog/2013/12/20/the-4-keys-to-100-uptime-with-nodejs）。William 有在 Node 上运行 Fluencia 和 SpanishDict 的经验，所以他是这方面的权威，并且他认为用域是保持 Node 正常运行的根本。Node 关于域的官方文档（http://nodejs.org/api/domain.html）也值得通读。

12.3.3　用多台服务器扩展

用集群向外扩展可以实现单台服务器的性能最大化，但当你需要多台服务器时会怎样？这时情况会变得有点复杂。要实现这种并行，你需要一台代理服务器（为了跟一般用于访问外部网络的代理区别开，经常被称为反向代理或正向代理，但我发现这种叫法既费解又没必要，所以我只称它为代理）。

在代理领域的两个后起之秀分别是 Nginx（读作"engine X"）和 HAProxy。Nginx 服务器简直像雨后春笋一般，我最近为公司做了一个竞争性分析，发现超过 80% 的竞争对手用的是 Nginx。Nginx 和 HAproxy 都是健壮、高性能的代理服务器，都能够胜任大多数苛刻的应用（如果你心存怀疑，可以参考一下 Netflix，它所占的互联网流量高达 30%，用了 Nginx）。

还有一些比较小的基于 Node 的代理服务器，比如 proxy（https://npmjs.org/package/proxy）和 node-http-proxy（https://www.npmjs.org/package/http-proxy）。如果你要求不高，或者是用于开发，这些都是很好的选择。对于生产环境而言，我推荐你用 Nginx 或 HAProxy（这两个都是免费的，尽管提供服务是收费的）。

安装和配置代理服务器超出了本书的范围，但它并不像你想得那么难（特别是如果你用 proxy 或 node-http-proxy）。目前使用集群已经可以保证我们的网站能向外扩展了。

如果你确实配置了一台代理服务器，请确保告知 Express 你用了代理，并且它应该得到信任：

```
app.enable('trust proxy');
```

这样可以确保 `req.ip`、`req.protocol` 和 `req.secure` 能反映客户端和代理服务器之间连接的细节，而不是客户端和你的应用之间的。还有，`req.ips` 将会是一个数组，表明原始客户端 IP 和所有中间代理的名称或 IP 地址。

12.4 网站监控

网站监控是你可以采取的最重要的（也是最常被忽视的）QA 措施之一。唯一一件比凌晨 3 点起床修复坏掉的网站更糟的事，是凌晨 3 点因为网站宕掉被老板叫起来（或者，同样糟糕的是早上到单位之后才意识到你的客户损失了上万美元的销售额，因为网站宕了一夜都没人发现）。

你对故障无能为力：它们就像死亡和税收一样不可避免。然而，唯一能让你的老板和客户信服你的工作很优秀的办法，就是总能比他们早知道发生故障了。

12.4.1 第三方正常运行监控

在网站服务器上正常运行一个监控就好像在一栋没人住的房子里装一个烟雾报警器。它可能可以发现某些页面不能访问了，但如果整个服务器都宕掉了，它甚至可能都发不出一个 SOS。所以你的第一道防线应该是第三方正常运行监控。UptimeRobot（http://uptimerobot.com/）有 50 个免费监控，并且配置简单。警报可以通过邮件、短信（文本消息）、Twitter 或者 iPhone 应用程序发送。你可以监控单个页面的返回码（除 200 之外的所有返回码都可以视为错误），或者检查页面上有没有某个关键字。不过要记住，如果你用关键字监控，它可能会影响你的分析（你可以从大多数分析服务中去掉正常运行监控产生的流量）。

如果你需要更精密的监控，还有其他更昂贵的服务，比如 Pingdom（http://pingdom.com/）和 Site24x7（http://www.site24x7.com/zhcn/index.html）。

12.4.2 应用程序故障

正常运行监控可以非常有效地监测大规模故障。如果你用关键字监控，它们甚至可以用来监测应用程序故障。比如，如果你在网站报告错误时笃定地包括关键字“服务器故障”，关键字监控就符合你的需要。然而，一般你在处理故障时都想表现得更优雅。给用户显示一个友好的消息“对不起，这项服务目前不正常”，并且你会收到一封邮件或一条短信告诉你有故障了。当你依赖第三方组件时，比如数据库或其他 Web 服务器，一般会采取这种方式。

一种简单的故障处理方式是有错误时给你自己发邮件。在第 11 章，我们展示了如何创建一个错误处理机制，使它在有错误时通知你。

如果通知需求复杂（比如，你有庞大的 IT 部门，其中一些人是轮班的“随叫随到”），你可能要考虑找一个通知服务，比如亚马逊的简单通知服务（SNS）。

> 你还可以看看专用的错误监控服务，比如 Sentry（https://getsentry.com/）或 Airbrake（https://airbrake.io/），它们提供的体验比收到错误通知邮件更友善。

12.5　压力测试

压力测试（或负载测试）是为了让你相信服务器可以正常地应对成百上千的并发请求。这也是可以独立成书的大课题，压力测试可能非常复杂，并且你想要它多复杂在很大程度上取决于你的项目。如果你有理由相信自己的网站可能非常受欢迎，可能要在压力测试上投入更多的时间。

现在我们先添加一个简单的测试，确保程序可以满足一秒内对主页的 100 次请求。我们用 Node 模块 loadtest 做压力测试：

```
npm install --save loadtest
```

接下来添加测试包，qa/tests-stress.js：

```
var loadtest = require('loadtest');
var expect = require('chai').expect;

suite('Stress tests', function(){
      test('Homepage should handle 100 requests in a second', function(done){
          var options = {
              url: 'http://localhost:3000',
              concurrency: 4,
              maxRequests: 100
          };
          loadtest.loadTest(options, function(err,result){
            expect(!err);
            expect(result.totalTimeSeconds < 1);
            done();
          });
      });

});
```

我们已经在 Grunt 中配置好 Mocha 任务了，所以只要运行 `grunt` 应该就能看到新的测试通过了（不要忘记首先在另一个窗口中启动服务器）。

第 13 章 持久化

所有网站和 Web 应用程序（除了最简单的）都需要某种持久化方式，即某种比易失性内存更持久的数据存储方式，这样当遇到服务器宕机、断电、升级和迁移等情况时数据才能保存下来。本章会讨论可用的持久化选择，重点是文档数据库。

13.1 文件系统持久化

实现持久化的一种方式是将数据存到扁平文件中（“扁平”的意思是文件没有内在结构，只是一串字节）。Node 通过 `fs`（文件系统）模块实现文件系统持久化。

文件系统持久化有一些不足之处，特别是它的扩展性不好。当你需要不止一台服务器以满足流量的需求时，除非所有服务器都能访问一个共享的文件系统，否则就会遇到文件系统持久化的问题。此外，因为扁平文件没有内在结构，定位、排序和过滤数据就变成了应用程序的负担。出于这些原因，你应该用数据库而不是文件系统来做数据排序。排序二进制文件是个例外，比如图片、音频文件或视频。尽管很多数据库可以处理这类数据，但极少能达到文件系统那种效率（尽管关于二进制文件的信息一般会存在数据库里，以便搜索、排序和过滤）。

如果你确实需要存储二进制数据，记得文件系统依然有扩展性不好的问题。如果你的主机不能访问共享的文件系统（一般是这样），你应该考虑将二进制文件存在数据库中（一般要做些配置，以免数据库被拖垮），或者基于云的存储服务，比如亚马逊 S3 或者微软 Azure 存储。

现在我们已经知道需要注意的问题了，接下来看看 Node 对文件系统的支持。我们会重温

第 8 章假期摄影大赛那个例子。在程序主文件中填上处理那个表单的处理器：

```
// 确保存在目录 data
var dataDir = __dirname + '/data';
var vacationPhotoDir = dataDir + '/vacation-photo';
fs.existsSync(dataDir) || fs.mkdirSync(dataDir);
fs.existsSync(vacationPhotoDir) || fs.mkdirSync(vacationPhotoDir);

function saveContestEntry(contestName, email, year, month, photoPath){
    // TODO……这个稍后再做
}

app.post('/contest/vacation-photo/:year/:month', function(req, res){
    var form = new formidable.IncomingForm();
    form.parse(req, function(err, fields, files){
        if(err) return res.redirect(303, '/error');
        if(err) {
            res.session.flash = {
                type: 'danger',
                intro: 'Oops!',
                message: 'There was an error processing your submission. ' +
                    'Pelase try again.',
            };
            return res.redirect(303, '/contest/vacation-photo');
        }
        var photo = files.photo;
        var dir = vacationPhotoDir + '/' + Date.now();
        var path = dir + '/' + photo.name;
        fs.mkdirSync(dir);
        fs.renameSync(photo.path, dir + '/' + photo.name);
        saveContestEntry('vacation-photo', fields.email,
            req.params.year, req.params.month, path);
        req.session.flash = {
            type: 'success',
            intro: 'Good luck!',
            message: 'You have been entered into the contest.',
        };
        return res.redirect(303, '/contest/vacation-photo/entries');
    });
});
```

这个内容有点多，我们把它分解一下。首先，我们创建了一个目录来存放上传的文件（如果它还不存在的话）。你可能想把 data 目录添加到 .gitignore 文件中，以免不慎把上传的文件提交到代码库里。然后创建了一个 Formidable 的 `IncomingForm` 实例，并调用它的 `parse` 方法，传入 `req` 对象。回调函数提供了所有的表单域和上传的文件。因为我们称上传域为 `photo`，所以会有个 `files.photo` 对象包含上传文件的信息。因为要防止冲突，所以我们不能用原来的文件名（比如两个用户都上传了 portland.jpg）。要避免这个问题，我们根据时间戳创建一个唯一目录，因为不太可能有两个用户在同一毫秒内都上传名为 portland.jpg 的文件。然后我们重命名（移动）上传的文件（Formidable 会给它一个临时文件名，可以从 `path` 属性中得到）为我们指定的文件名。

最后，我们需要某种方式将用户上传的文件跟他们的邮件地址（以及提交的年和月）关联起来。我们可以把这个信息编码到文件或目录名中，但我们倾向于将这一信息存在数据库里。因为我们还没学会怎么做，所以准备把这一功能封装在函数 `vacationPhotoContest` 中，在本章的后面部分再完成这个函数。

> 一般来说，你不应该信任用户上传的任何东西，因为它可能是攻击你网站的载体。比如，一个恶意用户可能轻易地将一个有害的可执行文件重命名为 .jpg 文件，然后上传，作为攻击的第一步（然后再找办法执行它）。同样，我们用浏览器提供的 `name` 属性命名这个文件也是有风险的，有些人可能会在文件名中插入一些特殊字符来滥用它。要让这段代码完全安全，我们会给这个文件一个随机名，只接受扩展名（确保它仅由字母数字字符组成）。

13.2　云持久化

云存储越来越流行了，我强烈建议你利用这些便宜又好用的服务。这里有一个将文件保存到亚马逊 S3 账号中的例子，看看多容易吧：

```
var filename = 'customerUpload.jpg';

aws.putObject({
    ACL: 'private',
    Bucket: 'uploads',
    Key: filename,
    Body: fs.readFileSync(__dirname + '/tmp/ + filename)
});
```

要了解更多信息，请查阅 AWS SDK 文档（http://aws.amazon.com/sdkfornodejs）。

还有一个用微软 Azure 完成相同任务的例子：

```
var filename = 'customerUpload.jpg';

var blobService = azure.createBlobService();
blobService.putBlockBlobFromFile('uploads', filename, __dirname +
    '/tmp/' + filename);
```

要了解更多信息，请查阅微软 Azure 文档（http://azure.microsoft.com/zh-cn/develop/nodejs/）。

13.3　数据库持久化

所有网站和 Web 应用程序（除了最简单的）都需要数据库。即便你的数据是二进制的，并且你用共享的文件系统或云存储，你也很有可能需要一个数据库来做那些二进制数据的目录。

依照传统，“数据库”是“关系型数据管理系统”（RDBMS）的简称。关系型数据库，比如 Oracle、MySQL、PostgreSQL 或 SQL Server，基于几十年的研究和正规的数据库原理。

现在它是一种十分成熟的技术，这些数据库的能量是毋庸置疑的。然而，除非是亚马逊或 Facebook，否则要扩展数据库由什么组成的想法太奢侈了。最近几年兴起了 NoSQL 数据库，它们正在挑战互联网数据存储现状。

如果宣称 NoSQL 数据库在某种程度上比关系型数据库强是愚蠢的，但它们确实有些优势（反之亦然）。尽管在 Node 程序中集成关系型数据库很容易，但看起来 NoSQL 几乎就是专为 Node 设计的。

两种最流行的 NoSQL 数据库是*文档数据库*和*键 – 值数据库*。文档数据库善于存储对象，这使得它们非常适合 Node 和 JavaScript。键 – 值数据库如其名所示，极其简单，对于数据模式可以轻松映射到键 – 值对的程序来说是很好的选择。

我觉得文档数据库代表了关系型数据库的限制和键 – 值数据库的简单性两者之间的最佳折中，因此我们会以文档数据库为例。MongoDB 是文档数据库中的佼佼者，现在也非常健壮和成熟。

13.3.1 关于性能

NoSQL 数据库的简单是一把双刃剑。认真规划一个关系型数据库是一项非常繁重的任务，但认真规划的好处是数据库能提供卓越的性能。不要愚蠢地以为，因为 NoSQL 数据库一般更简单，所以对它们调优以实现最佳性能就不是门艺术和科学了。

关系型数据库传统上依赖于它们严格的数据结构和几十年的优化研究而取得高性能。另一方面，NoSQL 数据库像 Node 一样，接受了互联网分布式的本性，专注于用并发来扩展性能（关系型数据库也支持并发，但一般只用于最有需要的应用程序）。

为数据库的性能和扩展性进行规划是一个大而复杂的课题，超出了本书的范围。如果你的应用程序需要高水平的数据库性能，我建议从 Kristina Chodorow 的《MongoDB 权威指南》（http://www.ituring.com.cn/book/1172）开始。

13.3.2 设置MongoDB

设置 MongoDB 实例的困难之处会随操作系统而变化。为了避开各种问题，我们选择免费的 MongoDB 托管服务 MongoLab。

> 除了 MongoLab 还有其他 MongoDB 服务，比如 MongoHQ，它也提供免费的开发 / 沙盒账号。但建议你不要将这些账号用于生产的目的。MongoLab 和 MongoHQ 都有为生产提供的账号，所以在做出选择之前，你应该先研究一下价格。等你往生产环境切换时，待在同一家托管服务提供商可以省去好多麻烦。

MongoLab 入手很简单。只要到 http://mongolab.com 上点击注册，填好注册表单，登录，然后你就到了个人主页。在数据库下面，你会看到“此时没有数据库”。点击“新建”，然后你就会进入新建数据库的页面，其中会有些选项要你选择。你首先要选的是云提供商。对于免费（沙盒）账号而言，选什么无关紧要，不过你应该找一个离你近的数据中心（然而并不是所有数据中心都提供沙盒账号）。选择“单节点（开发）”和沙盒。你可以选择自己要用的 MongoDB 版本，本书示例中用的是 2.4。最后，选择数据库名称，然后点击“新建 MongoDB 部署”。

13.3.3 Mongoose

尽管有底层的 MongoDB 驱动（https://npmjs.org/package/mongodb），但你可能还是想用对象文档映射（ODM）。有官方支持的 MongoDB ODM 是 Mongoose。

JavaScript 的优势之一是它的对象模型极其灵活。如果你想给一个对象添加属性或方法，尽管去做，并且不用担心要修改类。可惜，那种随心所欲的灵活性可能会对数据库产生负面影响，因为它们会变得零碎和难以调优。Mongoose 试图确立平衡，它引入了模式和模型（联合的，模式和模型类似于传统面向对象编程中的类）。模式很灵活，但仍为数据库提供了一些必要的结构。

在开始之前，我们要先把 Mongoose 模块装上：

```
npm install --save mongoose
```

然后将数据库凭证添加到 credentials.js 文件里：

```
mongo: {
    development: {
        connectionString: 'your_dev_connection_string',
    },
    production: {
        connectionString: 'your_production_connection_string',
    },
},
```

在 MongoLab 的数据库页面上有你的连接字符串，在你的个人主页上点击相应的数据库。你会在一个框里看到你的 MongoDB 连接 URI（以 mongodb:// 开头）。你还需要一个数据库用户。要创建用户，点击用户，然后“添加数据库用户”。

注意，我们存了两组凭证：一个用于开发，一个用于生产。你可以现在设置两个数据库，或者将两个指向同一个数据库（等正式启用的时候，你可以转换成使用两个单独的数据库）。

13.3.4 使用Mongoose连接数据库

我们先从创建数据库的连接开始：

```
var mongoose = require('mongoose');
var opts = {
    server: {
       socketOptions: { keepAlive: 1 }
    }
};
switch(app.get('env')){
    case 'development':
        mongoose.connect(credentials.mongo.development.connectionString, opts);
        break;
    case 'production':
        mongoose.connect(credentials.mongo.production.connectionString, opts);
        break;
    default:
        throw new Error('Unknown execution environment: ' + app.get('env'));
}
```

opts 对象是可选的，但我们想指定 keepAlive 选项，以防止长期运行的应用程序（比如网站）出现数据库连接错误。

13.3.5 创建模式和模型

接下来我们为草地鹨旅行社创建一个度假包数据库。先从定义模式和模型开始。创建文件 models/vacation.js：

```
var mongoose = require('mongoose');

var vacationSchema = mongoose.Schema({
    name: String,
    slug: String,
    category: String,
    sku: String,
    description: String,
    priceInCents: Number,
    tags: [String],
    inSeason: Boolean,
    available: Boolean,
    requiresWaiver: Boolean,
    maximumGuests: Number,
    notes: String,
    packagesSold: Number,
});
vacationSchema.methods.getDisplayPrice = function(){
    return '$' + (this.priceInCents / 100).toFixed(2);
};
var Vacation = mongoose.model('Vacation', vacationSchema);
module.exports = Vacation;
```

这段代码声明了 vacation 模型的属性，以及各个属性的类型。有几个字符串属性、两个数值属性、两个布尔属性，以及一个字符串数组（记为 [String]）。我们也可以在这里定义模式的方法。在存储产品价格时，我们用的单位是美分而不是美元，这样做是为了避开浮

点数的四舍五入问题，但很显然，在显示产品价格时我们肯定要按美元显示（当然，在国际化之前是这样的）。所以我们添加了方法 getDisplayPrice，以便得到恰当的价格显示。每个产品都有个“库存单位”（SKU），即便你觉得度假不是“库存产品”，但 SKU 是非常标准的会计概念，甚至不销售的有形货物也有这个属性。

只要有了模式，我们就可以用 mongoose.model 创建模型。从这点来看，Vacation 非常像传统的面向对象编程中的类。注意，在创建模型之前必须先定义方法。

> 由于浮点数的特质，在 JavaScript 中涉及金融计算时要谨慎。以美分为单位存储价格有帮助，但并不能根除这个问题。在下一版的 JavaScript（ES6）中会有个适合做金融计算的 decimal 类型。

我们输出了 Mongoose 创建的 Vacation 模型对象。要在程序中使用这个模型，我们可以像下面这样引入它：

```
var Vacation = require('./models/vacation.js');
```

13.3.6 添加初始数据

我们的数据库中还没有度假包，所以我们准备添加一些初始数据。你最终可能会创建一个管理产品的办法，但本书直接用代码来做了：

```
Vacation.find(function(err, vacations){
    if(vacations.length) return;

    new Vacation({
        name: 'Hood River Day Trip',
        slug: 'hood-river-day-trip',
        category: 'Day Trip',
        sku: 'HR199',
        description: 'Spend a day sailing on the Columbia and ' +
            'enjoying craft beers in Hood River!',
        priceInCents: 9995,
        tags: ['day trip', 'hood river', 'sailing', 'windsurfing', 'breweries'],
        inSeason: true,
        maximumGuests: 16,
        available: true,
        packagesSold: 0,
    }).save();

    new Vacation({
        name: 'Oregon Coast Getaway',
        slug: 'oregon-coast-getaway',
        category: 'Weekend Getaway',
        sku: 'OC39',
        description: 'Enjoy the ocean air and quaint coastal towns!',
        priceInCents: 269995,
```

```
        tags: ['weekend getaway', 'oregon coast', 'beachcombing'],
        inSeason: false,
        maximumGuests: 8,
        available: true,
        packagesSold: 0,
    }).save();

    new Vacation({
        name: 'Rock Climbing in Bend',
        slug: 'rock-climbing-in-bend',
        category: 'Adventure',
        sku: 'B99',
        description: 'Experience the thrill of climbing in the high desert.',
        priceInCents: 289995,
        tags: ['weekend getaway', 'bend', 'high desert', 'rock climbing'],
        inSeason: true,
        requiresWaiver: true,
        maximumGuests: 4,
        available: false,
        packagesSold: 0,
        notes: 'The tour guide is currently recovering from a skiing accident.',
    }).save();
});
```

这里用到了两个 Mongoose 方法。第一个是 `find`，如其名所示。在这个例子中，它会查找数据库中的所有 `Vacation` 实例，并将返回结果列表传给回调函数并调用。之所以这样做，是为了避免重复添加初始数据。如果数据库中已经有度假包了，那就是已经添加过了，我们可以快乐地走开了。然而在第一次执行时，`find` 返回的是空列表，所以我们继续创建两个度假产品，然后调用其上的 `save` 方法，将这些新对象保存到数据库中。

13.3.7　获取数据

我们已经见过 `find` 方法了，我们将会用它显示一个度假列表。然而这次我们准备传给 `find` 一个选项来过滤数据。具体来说，我们只想显示目前能够提供的度假产品。

给产品页创建个视图，views/vacations.handlebars：

```
<h1>Vacations</h1>
{{#each vacations}}
    <div class="vacation">
        <h3>{{name}}</h3>
        <p>{{description}}</p>
        {{#if inSeason}}
            <span class="price">{{price}}</span>
            <a href="/cart/add?sku={{sku}}" class="btn btn-default">Buy Now!</a>
        {{else}}
            <span class="outOfSeason">We're sorry, this vacation is currently
            not in season.
            {{! The "notify me when this vacation is in season"
                page will be our next task. }}
```

```
                <a href="/notify-me-when-in-season?sku={{sku}}">Notify me when
                this vacation is in season.</a>
            {{/if}}
        </div>
    {{/each}}
```

现在我们可以创建路由处理器把它全串起来：

```
// 参见配套源码库中的 /cart/add 路由……

app.get('/vacations', function(req, res){
    Vacation.find({ available: true }, function(err, vacations){
        var context = {
            vacations: vacations.map(function(vacation){
                return {
                    sku: vacation.sku,
                    name: vacation.name,
                    description: vacation.description,
                    price: vacation.getDisplayPrice(),
                    inSeason: vacation.inSeason,
                }
            })
        };
        res.render('vacations', context);
    });
});
```

这段代码大部分看起来应该挺熟悉的，但有些地方可能会令你吃惊。比如，我们处理度假列表视图上下文的方式看起来可能比较怪异。我们为什么要将从数据库里返回来的产品映射为几乎一样的对象？其中一个原因是 Handlebars 视图无法在表达式中使用函数的输出。所以为了以一个整齐的格式化方式显示价格，我们必须将其转为简单的字符串属性。我们可以这样做：

```
var context = {
    vacations: products.map(function(vacations){
        vacation.price = vacation.getDisplayPrice();
        return vacation;
    });
};
```

这当然可以省几行代码，但按我的经验，不要将未映射的数据库对象直接传给视图。视图会得到一堆它可能不需要的属性，并且可能是以它不能兼容的格式。到目前为止，我们的例子都很简单，但一旦它开始变得更复杂，你可能想要对传给视图的数据做更多定制化的处理。这样还很容易暴露机密信息，或者威胁网站安全的信息。因此我建议将数据库中返回的数据映射一下，并且只传递视图需要的数据（做必要的转换，就像我们对价格做的处理一样）。

在某些 MVC 架构的变体中，引入了一种称为“视图模型”的组件。视图模型本质上就是对模型的抽取和转换，从而让模型（或多个模型）更适合在视图中显示。我们上面做的基本上就是即时创建一个视图模型。

13.3.8 添加数据

我们已经知道如何添加数据了（在添加度假集合时添加了数据），也知道如何更新数据（当预定度假时我们更新了已销售包的数量），但接下来我们要看一个稍微有点复杂的场景，该场景凸显了文档数据库的灵活性。

当度假过季时，我们要显示一个链接，邀请客户在度假重新变得应季时接收通知。我们要实现这个功能，首先要创建模式和模型（models/vacationInSeasonListener.js）：

```
var mongoose = require('mongoose');

var vacationInSeasonListenerSchema = mongoose.Schema({
    email: String,
    skus: [String],
});
var VacationInSeasonListener = mongoose.model('VacationInSeasonListener',
    vacationInSeasonListenerSchema);

module.exports = VacationInSeasonListener;
```

然后创建视图，views/notify-me-when-in-season.handlebars：

```
<div class="formContainer">
    <form class="form-horizontal newsletterForm" role="form"
            action="/notify-me-when-in-season" method="POST">
        <input type="hidden" name="sku" value="{{sku}}">
        <div class="form-group">
          <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
          <div class="col-sm-4">
              <input type="email" class="form-control" required
                  id="fieldName" name="email">
          </div>
        </div>
        <div class="form-group">
            <div class="col-sm-offset-2 col-sm-4">
                <button type="submit" class="btn btn-default">Submit</button>
            </div>
        </div>
    </form>
</div>
```

最后是路由处理器：

```
var VacationInSeasonListener = require('./models/vacationInSeasonListener.js');
```

```
app.get('/notify-me-when-in-season', function(req, res){
    res.render('notify-me-when-in-season', { sku: req.query.sku });
});

app.post('/notify-me-when-in-season', function(req, res){
    VacationInSeasonListener.update(
        { email: req.body.email },
        { $push: { skus: req.body.sku } },
        { upsert: true },
        function(err){
            if(err) {
                console.error(err.stack);
                req.session.flash = {
                    type: 'danger',
                    intro: 'Ooops!',
                    message: 'There was an error processing your request.',
                };
                return res.redirect(303, '/vacations');
            }
            req.session.flash = {
                type: 'success',
                intro: 'Thank you!',
                message: 'You will be notified when this vacation is in season.',
            };
            return res.redirect(303, '/vacations');
        }
    );
});
```

这是什么魔法？我们怎么能在 `VacationInSeasonListener` 还不存在的时候更新其中的记录呢？答案在于 Mongoose 方便的 `upsert`（“更新”和“插入”的混成词）。基本上就相当于，如果给定邮件地址的记录不存在，就会创建它。如果记录存在，就更新它。然后我们用魔法变量 `$push` 表明我们想添加一个值到数组中。希望你能体会到 Mongoose 给你提供了什么，以及你为什么要用它而不是底层的 MongoDB 驱动。

> 如果用户多次填写表单，这段代码不能防止添加多个 SKU。当度假变得应季时，我们找出所有想要收到通知的客户，必须注意不要多次通知他们。

13.3.9 用MongoDB存储会话数据

我们在第 9 章讨论过，用内存存储会话数据不适用于生产环境。好在设置 MongoDB 用来存储会话非常容易。

我们会用 `session-mongoose` 包提供 MongoDB 会话存储。只要装上它（`npm install --save session-mongoose`），我们就可以在主程序文件中设置它：

```
var MongoSessionStore = require('session-mongoose')(require('connect'));
var sessionStore = new MongoSessionStore({ url:
    credentials.mongo.connectionString });

app.use(require('cookie-parser')(credentials.cookieSecret));
app.use(require('express-session')({ store: sessionStore }));
```

接下来我们要用新创建的会话存储做些有意义的事情。比如我们想要用不同的币种显示度假产品的价格。此外，我们还希望网站记住用户偏好的币种。

我们先要在度假产品页面底部添加一个币种选择器：

```
<hr>
<p>Currency:
    <a href="/set-currency/USD" class="currency {{currencyUSD}}">USD</a> |
    <a href="/set-currency/GBP" class="currency {{currencyGBP}}">GBP</a> |
    <a href="/set-currency/BTC" class="currency {{currencyBTC}}">BTC</a>
</p>
```

然后是一点 CSS：

```
a.currency {
    text-decoration: none;
}
.currency.selected {
    font-weight: bold;
    font-size: 150%;
}
```

最后我们会添加路由处理器来设定币种，并修改 /vacations 的路由处理器来用当前币种显示价格：

```
app.get('/set-currency/:currency', function(req,res){
    req.session.currency = req.params.currency;
    return res.redirect(303, '/vacations');
});

function convertFromUSD(value, currency){
    switch(currency){
        case 'USD': return value * 1;
        case 'GBP': return value * 0.6;
        case 'BTC': return value * 0.0023707918444761;
        default: return NaN;
    }
}

app.get('/vacations', function(req, res){
    Vacation.find({ available: true }, function(err, vacations){
        var currency = req.session.currency || 'USD';
        var context = {
            currency: currency,
            vacations: vacations.map(function(vacation){
```

```
                return {
                    sku: vacation.sku,
                    name: vacation.name,
                    description: vacation.description,
                    inSeason: vacation.inSeason,
                    price: convertFromUSD(vacation.priceInCents/100, currency),
                    qty: vacation.qty,
               }
           })
       };
       switch(currency){
           case 'USD': context.currencyUSD = 'selected'; break;
           case 'GBP': context.currencyGBP = 'selected'; break;
           case 'BTC': context.currencyBTC = 'selected'; break;
        }
        res.render('vacations', context);
    });
});
```

当然，这不是执行汇率换算的好办法，我们应该利用第三方汇率换算 API，以便确保汇率是最新的。但对于演示而言，这样就足够了。现在你可以在多种币种之间切换，去试试吧，重启服务器……你会发现它记住了你的币种偏好。如果你清除 cookie，币种偏好将会被忘记。你会看到我们漂亮的币种格式不见了，现在问题更复杂了，不过我把这个作为练习留给读者。

如果你看下数据库，会发现有个新集合“sessions”。如果你看看那个集合，会发现一个有你会话 ID 的文档（属性 `sid`）和你的币种偏好。

> MongoDB 不一定是会话存储的最佳选择，它有点杀鸡用牛刀的意味。另外一个流行又易用的会话持久化方案是用 Redis（http://redis.io/）。请参阅 `connect-redis` 包（https://www.npmjs.org/package/connect-redis）来了解如何设置使用 Redis 做会话存储。

第 14 章

路由

路由是网站或 Web 服务中最重要的一个方面；好在 Express 中的路由简单、灵活、健壮。路由是将请求（由 URL 和 HTTP 方法指定）路由到处理它们的代码去的一种机制。就像我们说过的，路由过去是基于文件的，并且非常简单：如果把文件 foo/about.html 放到网站上，你就可以通过路径 /foo/about.html 用浏览器访问它。这很简单，但不灵活。并且，如果你没注意到，现在如果 URL 中还有 HTML 就太落伍了。

在探讨 Express 路由技术之前，我们应该讨论下信息架构（IA）的概念。IA 是指内容的概念性组织。在考虑路由之前有一个可扩展（但不过于复杂的）IA 会为你的后续工作提供巨大的好处。

在关于 IA 的文章中，最有智慧也最经典的是 Tim Berners-Lee 写的，就是他发明了互联网。你现在可以（也应该）看看：http://www.w3.org/ Provider/Style/URI.html。这是他于 1998 年写的。先沉淀一分钟：1998 年的互联网技术还不像今天这样真实可见，这篇文章就是在当时那种情况下写成的。

这篇文章要求我们承担下面这个崇高的责任：

> 网站管理员有责任让分配的 URI 保持 2 年、20 年、200 年不变。这需要思考、组织和决心。
>
> ——Tim Berners-Lee

我想如果 Web 设计师像其他工程类职业一样，也要求有职业许可，那么我们会宣誓的。（细心的读者会发现一个幽默的事实，那篇文章的 URL 是以“.html”结尾的。）

打个比方（比较年轻的受众可能看不懂），想象每隔两年，你喜欢的图书馆就要完全重排杜威十进制系统。

有一天，你走进图书馆，发现自己什么也找不到。如果你重新设计 URL 结构，情况也会是一样。

认真思考你的 URL：它们在 20 年后还有意义吗？（200 年可能有点长：谁知道我们那时候还用不用 URL 呢。不过我佩服考虑得那么长远的精神。）认真考虑内容的分解。按逻辑归类，尽量别把自己逼入死角。这是科学，但也是艺术。

可能最重要的是跟其他人合作设计你的 URL。即便你是方圆几公里内最好的信息架构师，可能也会惊异地看到人们对相同内容的观点有多么不同。我的意思不是让你做一个从每个人的观点来看都有意义的 IA（因为那一般是不可能的），而是说以多种观点看待问题能让你产生更好的想法，并且暴露你自己的 IA 中的缺陷。

这里有些建议能帮你实现持久的 IA。

- *绝不在 URL 中暴露技术细节*
  你有没有过这种经历：看到 URL 以“.asp”结尾的网站，然后觉得那个网站过时到无可救药的地步了？记住，曾几何时，ASP 是前沿技术。尽管说起来很痛苦，但 JavaScript、JSON、Node 和 Express 也会落得如此下场。希望很多很多年后才会如此，但时间对技术是无情的。

- *避免在 URL 中出现无意义的信息*
  认真考虑 URL 中的每个单词。如果它没有任何意义，就去掉它。比如说，当网站在 URL 中使用单词 home 时总会让我退缩。根路由就是首页。你不需要像 /home/directions 和 /home/contact 这样的 URL。

- *避免无谓的长 URL*
  在同等条件下，短的 URL 比长的 URL 好。然而你不应该为了缩短 URL 牺牲清晰性，或者 SEO。缩写很诱人，但要认真考虑：在你把它们固定到 URL 中之前，它们应该是非常常见和普遍的。

- *单词分隔符要保持一致*
  用连字符分隔单词的情况十分常见，而用下划线的情况不太多。一般认为连字符比下划线更美观，并且大多数 SEO 专家都建议用连字符。不管你选择用连字符还是用下划线，都要保持一致。

- *绝不要用空格或不可录入的字符*
  不要在 URL 中使用空格。它一般会被转换成加号（+），会引起困惑。很明显你应该避免使用不可录入的字符，并且我要提醒你，一定不要使用除字母、数字、破折号和下划

线之外的任何字符。用的时候你可能觉得很聪明，但“聪明”经受不住时间的检验。很明显，如果网站的受众用的不是英语，你可能会用非英文字符（会被转换成百分比码），但如果你要本地化，可能会觉得头疼。

- *在 URL 中用小写字母*

  这可能会引起争论：有些人觉得 URL 中用混合大小写不仅是可以接受的，还应该优先使用。我不想挑起这种争论，但我要指出小写的好处，它总能由代码自动生成。如果你曾经要遍历网站净化上千个链接，或者做字符串比较，就会支持这种说法。我个人感觉小写字母的 URL 更美观，但最终决定权在你。

14.1　路由和SEO

如果你想让网站是可发现的（大多数人都会这样做），那就要考虑 SEO，以及 URL 会如何影响它。特别是如果某些关键字特别重要并且有意义，就考虑把它变成 URL 的一部分。比如说，草地鹨旅行社提供了几个俄勒冈海岸度假产品：要确保这些度假产品有较高的搜索引擎排名，我们在标题、头部、主体和元描述中使用字符串“俄勒冈海岸”，并且 URL 以 /vacations/oregon-coast 打头。Manzanita 度假包能在 /vacations/oregon-coast/manzanita 中找到。如果为了缩短 URL 用 /vacations/manzanita，我们可能会失去宝贵的 SEO。

这就是说，不要为了提高排名而往 URL 中塞关键字，否则会失败的。比如说，将 Manzanita 度假 URL 改成 /vacations/oregon-coast-portland-and-hood-river/oregon-coast/manzanita，这样多说了一次“Oregon Coast”，还同时提到了关键字“Portland”和“Hood River”，这是执迷不悟。这和良好的 IA 背道而驰，并且很可能会事与愿违。

14.2　子域名

除了路径，子域名一般也是 URL 中用来路由请求的部分。子域名最好保留给程序中显著不同的部分，比如 REST API（api.meadowlarktravel.com）或管理界面（admin.meadowlarktravel.com）。有时使用子域名是出于技术方面的原因。比如说，如果我们准备用 WordPress 搭建博客（而网站的其他部分用 Express），用 blog.meadowlarktravel.com 更容易（更好的方案是用代理服务器，比如 Nginx）。用子域名分割内容时一般会影响 SEO，所以一般应该留给 SEO 不重要的区域，比如管理区域和 API。记住这一点，并且只有在确实没有其他选择时，才给对于 SEO 方案来说比较重要的内容使用子域名。

Express 中的路由机制默认不会把子域名考虑在内：`app.get(/about)` 会处理对 http://meadowlarktravel.com/about、http:// www.meadowlarktravel.com/about 和 http://admin.meadowlarktravel.com/about 的请求。如果你想分开处理子域名，可以用 `vhost` 包（表示“虚拟主机”，源自 Apache 的机制，一般用来处理子域名）。先安装这个包（`npm install`

--save vhost)，然后编辑应用程序文件创建一个子域名：

```
// 创建子域名 "admin" ……它应该出现在所有其他路由之前
var admin = express.Router();
app.use(vhost('admin.*', admin));

// 创建 admin 的路由；它们可以在任何地方定义
admin.get('/', function(req, res){
        res.render('admin/home');
});
admin.get('/users', function(req, res){
        res.render('admin/users');
});
```

express.Router() 本质上是创建了一个新的 Express 路由器实例。你可以像对待原始实例（app）那样对它：像对 app 那样给它添加路由和中间件。然而在将它添加到 app 上之前，它什么也不会做。我们通过 vhost 添加它，将那个路由器实例绑到那个子域名。

14.3　路由处理器是中间件

我们已经见过非常基本的路由了：只是匹配给定的路径。但 app.get('/foo',...) 究竟做了什么呢？如第 10 章所述，它只是一种特殊的中间件，向下会有一个 next 方法传入。我们来看几个更复杂的例子。

```
app.get('/foo', function(req,res,next){
        if(Math.random() < 0.5) return next();
        res.send('sometimes this');
});
app.get('/foo', function(req,res){
        res.send('and sometimes that');
});
```

在这个例子中，同一个路由有两个处理器。一般第一个会赢，但这样第一个会有大概一半的机会只是经过，将机会留给第二个处理器。我们甚至不需要用两次 app.get：可以在一个 app.get 使用任意多个处理器。在下面这个例子中，三种不同的响应出现的几率差不多：

```
app.get('/foo',
        function(req,res, next){
                if(Math.random() < 0.33) return next();
                res.send('red');
        },
        function(req,res, next){
                if(Math.random() < 0.5) return next();
                res.send('green');
        },
        function(req,res){
                res.send('blue');
        },
)
```

尽管乍一看可能不是特别实用，但这让你可以创建可以用在任何路由中的通用函数。比如说，我们有种机制在特定页面上显示特殊优惠。特殊优惠经常换，并且不是每个页面上都显示。我们可以创建一个函数，将 specials 注入到 res.locals 属性中（第 7 章讲过）：

```
function specials(req, res, next){
        res.locals.specials = getSpecialsFromDatabase();
        next();
}
app.get('/page-with-specials', specials, function(req,res){
        res.render('page-with-specials');
});
```

我们也可以用这种方式实现授权机制。比如说我们的用户授权代码会设定一个会话变量 req.session.authorized，则可以像下面这样做一个可重复使用的授权过滤器：

```
function authorize(req, res, next){
        if(req.session.authorized) return next();
        res.render('not-authorized');
}
app.get('/secret', authorize, function(){
        res.render('secret');
})
app.get('/sub-rosa', authorize, function(){
        res.render('sub-rosa');
});
```

14.4　路由路径和正则表达式

路由中指定的路径（比如 /foo）最终会被 Express 转换成一个正则表达式。某些正则表达式中的元字符可以用在路由路径中：+、?、*、(和)。我们看两个例子。比如你想用同一个路由处理 /user 和 /username 两个 URL：

```
app.get('/user(name)?', function(req,res){
        res.render('user');
});
```

http://khaaan.com 是我最喜欢的新奇网站之一。去吧：你去看的时候我会在这里等着的。感觉好点儿吗？好。假如我们想要做自己的“KHAAAAAAAAN”页面，但不想让用户记住是 2 个 a、3 个 a，还是 10 个 a。下面这段代码可以胜任这一任务：

```
app.get('/khaa+n', function(req,res){
        res.render('khaaan');
});
```

并不是所有的常规正则表达式元字符在路由路径中都有含义，虽然只有前面列出来的那些。这很重要，因为一般在正则表达式中表示“任意字符”的句号点（.）可以不经转义用在路由中。

最后，如果你的路由真的需要功能完整的正则表达式，也可以支持的：

```
app.get(/crazy|mad(ness)?|lunacy/, function(req,res){
        res.render('madness');
});
```

我还在寻找在路由路径中使用正则元字符的好理由，比完整的正则少很多，但还是要知道有这个功能的。

14.5 路由参数

在你日常使用的 Expression 工具箱中可能很少发现正则路由，但路由参数很可能要经常用。简而言之，这是一种把变量参数放到路由中成为其一部分的办法。比如我们想给每位职员一个页面。我们的数据库中有职员的简介和图片。随着公司规模的增长，给每位职员添加新的路由变得越来越不现实。我们看一下路由参数是怎么帮我们的：

```
var staff = {
        mitch: { bio: 'Mitch is the man to have at your back in a bar fight.' },
        madeline: { bio: 'Madeline is our Oregon expert.' },
        walt: { bio: 'Walt is our Oregon Coast expert.' },
};

app.get('/staff/:name', function(req, res){
        var info = staff[req.params.name];
        if(!info) return next();  // 最终将会落入 404
        res.render('staffer', info);
})
```

注意我们在路由中如何使用 :name。它会跟任何字符串匹配（不包括反斜杠），并将其跟键 `name` 一起放到 `req.params` 对象中。我们会经常用到这个参数，特别是在创建 REST API 时。路由中可以有多个参数。比如说，如果我们想按城市分解职员列表：

```
var staff = {
        portland: {
                mitch: { bio: 'Mitch is the man to have at your back.' },
                madeline: { bio: 'Madeline is our Oregon expert.' },
        },
        bend: {
                walt: { bio: 'Walt is our Oregon Coast expert.' },
        },
};

app.get('/staff/:city/:name', function(req, res){
        var info = staff[req.params.city][req.params.name];
        if(!info) return next(); // 最终将会落入 404
        res.render('staffer', info);
});
```

14.6 组织路由

你可能已经清楚了，在主应用程序文件中定义所有路由太笨重了。那样不仅会导致那个文件一直增长，还不利于功能的分离，因为那个文件里已经有很多东西了。一个简单的网站可能只有十几个路由，甚至更少，但比较大的网站可能有上百个路由。

那么如何组织路由呢？你想怎么组织自己的路由？ Express 对于你如何组织路由没有意见，所以怎么做完全是你的事情。

我会在下一节谈到处理路由的流行做法，但现在我要先推荐下面这四条组织路由的指导原则。

- *给路由处理器用命名函数*
  到目前为止，我们都是在行内写路由处理器的，实际上就是马上在那里定义处理路由的函数。这对于小程序或原型来说没问题，但随着网站的增长，这种方式很快就会变得过于笨重。

- *路由不应该神秘*
  这个原则故意说得比较模糊，因为大型的复杂网站可能比只有 10 个页面的网站需要更加复杂的组织方案。一种极端的做法是简单地把网站的所有路由都放到一个文件中，好知道它们在哪。对于大型网站来说，你可能不想这样，那就根据功能区域把路由分开。然而，即便如此，也应该清楚该到哪里找给定的路由。当你需要修订错误时，肯定不想花上一个小时来确定那个路由是在哪里处理的。我手头有一个 ASP.NET MVC 项目就有这种恐怖的问题：路由至少出现在 10 个不同的地方，并且毫无逻辑可言，也不一致，经常是自相矛盾的。即便我对那个（非常大的）网站非常熟悉，也要花好多时间追踪某个路由是在哪里处理的。

- *路由组织应该是可扩展的*
  如果你现在有 20 或 30 个路由，把它们都放在一个文件中可能没问题。如果在 3 年内你有了 200 个路由呢？这是有可能的。不管你选择用什么办法，都应该确保有增长的空间。

- *不要忽视自动化的基于视图的路由处理器*
  如果你的网站由很多静态和固定 URL 的页面组成，你的所有路由最终看起来将像是：`app.get('/static/thing', function(req, res){ res.render('static/thing'); }`。要减少不必要的重复代码，可以考虑使用自动化的基于视图的路由处理器。本章后面介绍了这种方式，并且它可以跟定制路由一起用。

14.7 在模块中声明路由

组织路由的第一步是把它们都放到它们自己的模块中。这有很多种办法。一种方式是将你

的模块做成一个函数，让它返回包含“方法”和“处理器”属性的对象数组。然后你可以这样在应用程序文件中定义路由：

```
var routes = require('./routes.js')();

routes.forEach(function(route){
        app[route.method](route.handler);
})
```

这种方式有它的优势，并且可能非常适合动态地存储路由，比如在数据库或 JSON 文件中。然而，如果你不需要那样的功能，我建议将 `app` 实例传给模块，然后让它添加路由。我们的例子中用的就是这种方式。创建文件 routes.js，将所有路由都放进去：

```
module.exports = function(app){

        app.get('/', function(req,res){
                app.render('home');
        }))

        //...

};
```

如果只是剪切粘贴，我们可能会遇到一些问题。比如说，我们的 `/about` 处理器用的 `fortune` 对象在这个上下文中没有。我们可以添加必要的引入，但先等一下：我们很快就要把处理器挪到它们自己的模块中去了，然后我们会解决这个问题。

那么我们如何连入路由呢？简单，在 meadowlark.js 中直接引入路由：

```
require('./routes.js')(app);
```

14.8　按逻辑对处理器分组

要满足第一条指导原则（给路由处理器用命名函数），我们需要找地方放那些处理器。更极端的做法是给每个处理器建一个 JavaScript 文件。我很难想象这种方式在哪种场景下会带来好处。以某种方式将相关功能分组更好。那样不仅更容易利用共享的功能，并且更容易修改相关的方法。

现在我们先把功能分组到各自的文件中：handlers/main.js 中放首页处理器、`/about` 处理器，以及所有不属于任何其他逻辑分组的处理器，handlers/vacations.js 中放跟度假相关的处理器，以此类推。

看一下 handlers/main.js：

```
var fortune = require('../lib/fortune.js');
```

```
exports.home = function(req, res){
        res.render('home');
};

exports.about = function(req, res){
        res.render('about', {
                fortune: fortune.getFortune(),
                pageTestScript: '/qa/tests-about.js'
        } );
};

//...
```

接下来修改 routes.js 以使用它：

```
var main = require('./handlers/main.js');

module.exports = function(app){

        app.get('/', main.home);
        app.get('/about', main.about);
        //...

};
```

这满足了所有的指导原则。/routes.js 非常直白。一眼就能看出来网站里有哪些路由，以及它们是在哪里处理的。我们还预留了充足的增长空间。我们可以把相关功能放到很多不同的文件中。如果 routes.js 变得笨重了，我们可以再用相同的技术，把 app 传给另一个模块，再注册更多路由（尽管这已经开始变得“过于复杂”了，确保你只在真的有那么复杂的时候才用这种方式！）。

14.9　自动化渲染视图

如果你希望回到以前，只要把 HTML 文件放到一个目录中，然后很快你的网站就能提供它的旧时光，那么有这样想法的人不止你一个。如果你的网站有很多内容，但功能不多，你可能发现给每个视图添加一个路由是不必要的麻烦。好在我们可以解决这个问题。

比如说你想添加文件 views/foo.handlebars，然后它就神奇地可以通过路由 /foo 访问了。我们看看怎么做。在我们的应用程序文件中，就在 404 处理器之前，添加下面的中间件：

```
var autoViews = {};
var fs = require('fs');

app.use(function(req,res,next){
    var path = req.path.toLowerCase();
    // 检查缓存；如果它在那里，渲染这个视图
    if(autoViews[path]) return res.render(autoViews[path]);
    // 如果它不在缓存里，那就看看有没有 .handlebars 文件能匹配
```

```
    if(fs.existsSync(__dirname + '/views' + path + '.handlebars')){
        autoViews[path] = path.replace(/^\//, '');
        return res.render(autoViews[path]);
    }
    // 没发现视图；转到 404 处理器
    next();
});
```

现在我们只要添加个 .handlebars 文件到 view 目录下，它就神奇地渲染在相应的路径上了。注意，常规路由会避开这一机制（因为我们把自动视图处理器放在了其他所有路由后面），所以如果你有个路由为 /foo 渲染了不同的视图，那它会取得优先权。

14.10 其他的路由组织方式

我发现我在这里列出来的方式在灵活性和工作量之间实现了很好的平衡。然而还有其他流行的路由组织方式。好消息是它们跟我在这里介绍的技术不冲突。所以如果你发现自己网站的某些领域用不同的组织方式能工作得更好，可以随意搭配这些技术（不过你要冒着让架构变得扑朔迷离的风险）。

最流行的两种路由组织方式是命名空间路由（namespaced routing）和随机应变路由（resourceful routing）。当很多路由都以相同的前缀开始时，命名空间路由很不错（比如 /vacations）。有个 Node 模块叫 `express-namespace`，它让这种方式变得很容易。随机应变路由基于一个对象中的方法自动添加路由。如果网站的逻辑是天然面向对象的，这项技术就很好用。`express-resource` 包是如何实现这种路由组织风格的范例。

路由在项目中很重要，如果我在本章中介绍的基于模块的路由技术看起来不适合你，我建议你看看 `express-namespace` 或 `express-resource` 的文档。

第15章

REST API和JSON

到目前为止我们一直在设计供浏览器访问的网站。现在我们将注意力转移到将数据和功能提供给其他程序上。渐渐地，互联网不再是各自为政的网站集合了，而是一个真正的网：网站为了给用户提供更丰富的体验而相互通信。程序员的梦想成真了：代码可以像真人那样访问互联网了。

本章将会给应用添加一个 Web 服务（Web 服务器和 Web 服务没有理由不能在一个应用程序中共存）。“Web 服务”是一个通用术语，指任何可以通过 HTTP 访问的应用程序编程界面（API）。Web 服务的想法已经出现相当长的时间了，但直到不久之前，那些实现 Web 服务的技术还是沉闷的、错乱的、过于复杂的。现在仍然有使用那些技术（比如 SOAP 和 WSDL）的系统，也有帮你与这些系统交互的 Node 包。不过我们不会讲到这些，相反，我们的重点是提供“REST 风格”的服务，与其交互要更直接得多。

缩略词 REST 表示“表述性状态传输”（Representational State Transfer），念起来有点麻烦的“REST 风格”作为一个形容词来形容满足 REST 原则的 Web 服务。REST 的正规描述很复杂，需要计算机科学形式上的表述，但 REST 基本上就是客户端和服务器端的无状态连接。REST 的正式定义还指出服务可以被缓存，可以被分层（即当你使用一个 REST API 时，可能还有其他 REST API 在它下面）。

从实用角度来看，因为 HTTP 的限制，实际上很难创建出非 REST 风格的 API；比如说，你需要自己想办法确立状态。所以我们的工作大部分是取出需要的部分。

我们将会添加一个 REST API 到草地鹨旅行社网站上。为了鼓励到俄勒冈旅游，草地鹨旅行社维护着一个景点数据库，并配以有趣的历史事实。API 允许创建移动端应用，让游客

可以用他们的手机或平板自我导游。如果设备能感知位置，应用还可以让他们知道自己是否靠近一个有趣的景点。为了让数据库增长，API 还支持添加地标和景点（会进入审批队列以防滥用）。

15.1　JSON和XML

提供 API 的关键是有相通的语言。通信部分已经决定了，我们必须用 HTTP 方法跟服务器通信。但在那之后，我们可以用任何数据语言。传统上 XML 是非常流行的选择，并且是很重要的标记语言。尽管 XML 不是特别复杂，但 Douglas Crockford 觉得还可以做得更轻量，因此 JavaScript 对象标记（JSON）诞生了。除了对 JavaScript 非常友善（但它绝不是专有的，它是任何语言都可以解析的简单格式），它还有个优势，即一般手写起来也比 XML 更容易。

相比 XML 而言，我在大多数应用程序中都更喜欢 JSON：有更好的 JavaScript 支持，并且它是简单紧凑的格式。我建议侧重于 JSON，并且只在已有系统要求用 XML 跟你的应用通信时才提供 XML。

15.2　我们的API

在实现之前，我们会先把 API 规划好。我们想要下面这些功能：

- `GET /api/attractions`
  获取景点。以 `lat`、`lng` 和 `radius` 为查询字符串参数，返回一个景点列表。

- `GET /api/attraction/:id`
  根据 ID 返回一处景点。

- `POST /api/attraction`
  以 `lat`、`lng`、`name`、`description` 和 `email` 为请求体添加新的景点。新添加的景点会进入一个待审批队列。

- `PUT /api/attraction/:id`
  更新一处已有的景点。参数为景点的 `ID`、`lat`、`lng`、`name`、`description` 和 `email`。更新会进入待审批队列。

- `DEL /api/attraction/:id`
  删除景点。参数为景点 `ID`、`email` 和 `reason`。删除会进入待审批队列。

我们可以有很多描述 API 的方式。不过这里选择用 HTTP 方法和路径的组合来区分 API 调用，并用查询字符串和请求主体参数混合的方式传递数据。作为选择，我们可以用方法全

都相同的不同路径（比如 /api/attractions/delete）。[1] 我们也可以用同一种方式传递数据。比如说，我们可以选择在 URL 中用查询参数而不是查询字符串传递所有必需的信息：`GET/api/attractions/:lat/:lng/:radius`。为了避免出现超长的 URL，我建议用请求主体传递大块数据（比如景点的描述）。

> 将 `POST` 用于创建而 `PUT` 用于更新（或修改），这已经成为标准了。这些单词的英文含义并不支持这种分别，所以你可能要考虑用路径来区分这两种操作以避免混淆。

为简便起见，我们只会实现其中三个功能：添加景点、获取单个景点和获取景点列表。如果你下载了本书配套的源码，可以看到完整的实现。

15.3　API错误报告

HTTP API 的错误报告一般是通过 HTTP 状态码实现的：如果返回的响应码是 200（OK），则客户端知道请求成功了；如果响应码是 500（服务器内部错误），则请求失败了。然而在大多数应用程序中，并不是所有事情都可以（或者应该）粗略地划分成“成功”或“失败”。比如说，你用 ID 请求某件东西，但如果那个 ID 不存在怎么办？这不是服务器错误：客户端请求了不存在的东西。一般来说，错误可以分为以下几类。

- *灾难性错误*

  导致服务器的状态不稳定或不可知的错误。这种错误一般是未处理异常导致的。从灾难性错误中恢复的唯一办法是重启服务器。理想情况下，所有挂起的请求都会收到响应码 500，但如果故障很严重，服务器可能根本无法响应，请求会超时。

- *可恢复的服务器错误*

  可恢复错误不需要服务器重启，或其他任何壮烈的动作。这种错误一般是服务器上未预料到的错误条件导致的（比如不可用的数据库连接）。问题可能是暂时的或永久的。这种情况下应该返回响应码 500。

- *客户端错误*

  客户端错误是客户端犯了错误，一般是参数漏掉了或参数无效。这时不应该用响应码 500，毕竟服务器没有故障。一切都正常，只是客户端没有正确使用 API。此时你有两个选择：可以用状态码 200，并在响应体中描述错误，或者你可以尝试额外用恰当的 HTTP 状态码描述错误。我建议用后一种方式。这种情况下最合适的响应码是 404（未

注 1：如果你的客户端不能使用不同的 HTTP 方法，请参阅 https://github.com/expressjs/method-override，它可以让你“模拟”不同的 HTTP 方法。

找到）、400（错误的请求）和 401（未授权）。此外，响应体中应该有错误具体情况的说明。如果你想做得更好，错误消息中甚至应该包含文档的链接。注意，如果用户请求的是一个列表，但没有东西返回，这不是错误：返回空列表是恰当的响应。

在我们的应用程序中，会用 HTTP 响应码和响应体中错误消息的组合。注意，这种方式兼容 jQuery，这其中很重要的是考虑到用 jQuery 访问 API 的情况非常盛行。

15.4 跨域资源共享

如果你发布了一个 API，应该很想让其他人能够访问这个 API。这会导致*跨站 HTTP 请求*。跨站 HTTP 请求一直是很多攻击的对象，因此受到了*同源策略*的限制，限制可以从哪里加载脚本。具体来说就是协议、域和端口必须匹配。这使得其他网站不可能使用你的 API，所以有了跨域资源共享（CORS）。CORS 允许你针对个案解除这个限制，甚至允许你列出具体哪些域可以访问这个脚本。CORS 是通过 `Access-Control-Allow-Origin` 响应头实现的。在 Express 程序中最容易的实现方式是用 `cors` 包（`npm install --save cors`）。要在程序中启用 CORS：

```
app.use(require('cors')());
```

基于同源 API 存在的原因（防止攻击），我建议只在必要时应用 CORS。就我们的情况而言，想要输出整个 API（但只有 API），所以要将 CORS 限制在以“/api”开头的路径上：

```
app.use('/api', require('cors')());
```

请参阅包文档（https://www.npmjs.org/package/cors）了解 CORS 更高级的用法。

15.5 我们的数据存储

我们再一次要用 Mongoose 给数据库中的景点模型创建模式。创建文件 models/attraction.js：

```
var mongoose = require('mongoose');

var attractionSchema = mongoose.Schema({
    name: String,
    description: String,
    location: { lat: Number, lng: Number },
    history: {
        event: String,
        notes: String,
        email: String,
        date: Date,
    },
    updateId: String,
    approved: Boolean,
});
```

```
var Attraction = mongoose.model('Attraction', attractionSchema);
module.exports = Attraction;
```

因为更新需要审批，所以不能让 API 直接更新原始记录。我们的办法是创建一个指向原始记录的新记录（在它的属性 `updateId` 中）。一旦这个记录得到批准，我们就可以用更新记录中的信息更新原始记录，并删除这条更新记录。

15.6　我们的测试

如果用了 `GET` 之外的 HTTP 动词，那 API 的测试可能是个麻烦，因为浏览器只知道如何发起 `GET` 请求（以及从表单发起 `POST` 请求）。这有解决办法，比如优秀的“Postman - REST Client”Chrome 插件。然而，不管你是否使用这样的工具，有自动化测试总是好的。在给 API 写测试之前，我们需要一种实际调用 REST API 的办法。为此要用到 Node 包 `restler`：

```
npm install --save-dev restler
```

我们准备在 qa/tests-api.js 中实现对 API 的测试：

```
var assert = require('chai').assert;
var http = require('http');
var rest = require('restler');

suite('API tests', function(){
    var attraction = {
        lat: 45.516011,
        lng: -122.682062,
        name: 'Portland Art Museum',
        description: 'Founded in 1892, the Portland Art Museum\'s colleciton ' +
            'of native art is not to be missed.  If modern art is more to your ' +
            'liking, there are six stories of modern art for your enjoyment.',
        email: 'test@meadowlarktravel.com',
    };

    var base = 'http://localhost:3000';

    test('should be able to add an attraction', function(done){
        rest.post(base+'/api/attraction', {data:attraction}).on('success',
                function(data){
            assert.match(data.id, /\w/, 'id must be set');
            done();
        });
    });

    test('should be able to retrieve an attraction', function(done){
        rest.post(base+'/api/attraction', {data:attraction}).on('success',
                function(data){
            rest.get(base+'/api/attraction/'+data.id).on('success',
                    function(data){
                assert(data.name===attraction.name);
```

```
                    assert(data.description===attraction.description);
                    done();
                })
            })
        });
    });
```

注意，对获取景点的测试中，我们先添加了一个景点。你可能觉得没必要这么做，因为第一个测试已经添加过了，但这样做有两个原因。第一个原因是实战性的：即便测试在文件中的出现顺序是那样的，但因为 JavaScript 的异步性，我们不能保证 API 的调用也按那个顺序执行。第二个原因是原则性的：所有测试都应该完全独立，不能相互依赖。

这段代码的语法应该很直白：调用 `rest.get` 或 `rest.put`，把 URL 传给它，以及一个有 `data` 属性的对象，用来做请求体。这个方法返回一个发起事件的 promise。我们感兴趣的是 `success` 事件。当你在应用程序中使用 `restler` 时，可能也想监听其他事件，比如 `fail`（服务器给出的响应状态码是 4xx）或 `error`（连接或解析错误）。请查阅 `restler` 文档（https://github.com/danwrong/restler）了解更多信息。

15.7　用Express提供API

Express 十分擅长提供 API。本章后面还会介绍如何用 Node 模块提供额外的功能，但现在先从纯粹的 Express 实现开始：

```
var Attraction = require('./models/attraction.js');

app.get('/api/attractions', function(req, res){
    Attraction.find({ approved: true }, function(err, attractions){
        if(err) return res.send(500, 'Error occurred: database error.');
        res.json(attractions.map(function(a){
            return {
                name: a.name,
                id: a._id,
                description: a.description,
                location: a.location,
            }
        }));
    });
});

app.post('/api/attraction', function(req, res){
    var a = new Attraction({
        name: req.body.name,
        description: req.body.description,
        location: { lat: req.body.lat, lng: req.body.lng },
        history: {
            event: 'created',
            email: req.body.email,
            date: new Date(),
```

```
        },
        approved: false,
    });
    a.save(function(err, a){
        if(err) return res.send(500, 'Error occurred: database error.');
        res.json({ id: a._id });
    });
});

app.get('/api/attraction/:id', function(req,res){
    Attraction.findById(req.params.id, function(err, a){
        if(err) return res.send(500, 'Error occurred: database error.');
        res.json({
            name: a.name,
            id: a._id,
            description: a.description,
            location: a.location,
        });
    });
});
```

注意，在返回景点时，我们不是直接返回从数据库中返回来的模型。那样会暴露内部实现细节。相反，我们选出所需信息构造了一个新的对象返回。

如果现在运行测试（用 Grunt 或 `mocha -u tdd -R spec qa/tests-api.js`），应该能看到测试通过了。

15.8 使用REST插件

如你所见，只用 Express 写 API 很容易。然而用 REST 插件有些优势。接下来我们用健壮的 `connect-rest` 让 API 可以面向未来。先装上它：

```
npm install --save connect-rest
```

然后在 meadowlark.js 中引入它：

```
var rest = require('connect-rest');
```

API 不应该跟网站的常规路由冲突（确保你没有创建任何以“/api”开头的网站路由）。我建议把 API 路由放在网站路由后面：`connect-rest` 模块会检查每一个请求，向请求对象中添加属性，还会做额外的日志记录。因此把它放在网站路由后面更好，但要在 404 处理器之前：

```
// 网站路由在这里

// 在这里用 rest.VERB 定义 API 路由……

// API 配置
```

```
var apiOptions = {
    context: '/api',
    domain: require('domain').create(),
};

// 将 API 连入管道
app.use(rest.rester(apiOptions));

// 404 处理器在这里
```

> 如果你想最大化地分离网站和 API，可以考虑用子域名，比如 api.meadowlark.com。稍后我们会看到一个这样的例子。

`connect-rest` 已经提高了一点效率：我们可以自动给所有 API 调用加上前缀“/api”。这减少了手误的几率，并且可以在需要时轻松修改根 URL。

现在看一下如何添加 API 方法：

```
rest.get('/attractions', function(req, content, cb){
    Attraction.find({ approved: true }, function(err, attractions){
        if(err) return cb({ error: 'Internal error.' });
        cb(null, attractions.map(function(a){
            return {
                name: a.name,
                description: a.description,
                location: a.location,
            };
        }));
    });
});

rest.post('/attraction', function(req, content, cb){
    var a = new Attraction({
        name: req.body.name,
        description: req.body.description,
        location: { lat: req.body.lat, lng: req.body.lng },
        history: {
            event: 'created',
            email: req.body.email,
            date: new Date(),
        },
        approved: false,
    });
    a.save(function(err, a){
        if(err) return cb({ error: 'Unable to add attraction.' });
        cb(null, { id: a._id });
    });
});

rest.get('/attraction/:id', function(req, content, cb){
```

```
        Attraction.findById(req.params.id, function(err, a){
            if(err) return cb({ error: 'Unable to retrieve attraction.' });
            cb(null, {
                name: attraction.name,
                description: attraction.description,
                location: attraction.location,
            });
        });
    });
```

REST 函数不是只有常见的请求 / 响应两个参数，而是有三个：一个请求（跟平常一样）；一个内容对象，是请求被解析的主体；一个回调函数，可以用于异步 API 的调用。因为我们用了数据库，这是异步的，所以必须用回调将响应发给客户端（也有同步 API，你可以在 connect-rest 文档中看到：https://github.com/imrefazekas/connect-rest）。

注意，我们在创建 API 时还指定了一个域（见第 12 章）。这样我们可以孤立 API 错误并采取相应的行动。当在那个域中检测到错误时，connect-rest 会自动发送一个响应码 500，你所要做的只是记录日志并关闭服务器。比如：

```
apiOptions.domain.on('error', function(err){
    console.log('API domain error.\n', err.stack);
    setTimeout(function(){
        console.log('Server shutting down after API domain error.');
        process.exit(1);
    }, 5000);
    server.close();
    var worker = require('cluster').worker;
    if(worker) worker.disconnect();
});
```

15.9 使用子域名

因为 API 实质上是不同于网站的，所以很多人都会选择用子域将 API 跟网站其余部分分开。这十分容易，我们重构这个例子，将 meadowlarktravel.com/api 改成用 api.meadowlarktravel.com。

首先确保 vhost 中间件已经装好了（npm install --save vhost）。在开发环境中，你可能没有自己的域名服务器（DNS），所以我们需要用一种手段让 Express 相信你连接了一个子域。为此需要向 hosts 文件中添加一条记录。在 Linux 和 OS X 系统中，hosts 文件是 /etc/hosts；在 Windows 中是 %SystemRoot%\system32\drivers\etc\hosts。如果测试服务器的 IP 地址是 192.168.0.100，则在 hosts 文件中添加下面这行记录：

```
192.168.0.100   api.meadowlark
```

如果你是直接在开发服务器上工作，可以用 127.0.0.1（相当于本地服务器）代替真实的 IP 地址。现在我们直接连入新的 vhost 创建子域：

```
app.use(vhost('api.*', rest.rester(apiOptions));
```

还需要修改上下文：

```
var apiOptions = {
    context: '/',
    domain: require('domain').create(),
};
```

全都在这里了。现在所有通过 `rest.VERB` 定义的 API 路由都可以在 api 子域上调用了。

第 16 章

静态内容

静态内容是指应用程序不会基于每个请求而去改变的资源。下面这些一般都应该是静态内容。

- 多媒体

  图片、视频和音频文件。当然，图片很有可能是即时生成的（尽管不太常见，但视频和音频也有可能如此），但大多数多媒体资源都是静态的。

- CSS

  即便使用 LESS、Sass 或 Stylus 这样的抽象 CSS 语言，最后浏览器需要的还是普通 CSS。[1] 普通 CSS 是静态资源。

- JavaScript

  服务器端运行的是 JavaScript 并不意味着没有客户端 JavaScript。客户端 JavaScript 是静态资源。当然，现在界限开始变得有点儿模糊了：我们既想在后台使用，又想在客户端使用的通用代码算什么呢？这个问题有解决办法，但最终送到客户端的 JavaScript 通常是静态的。

- 二进制下载文件

  这包含所有种类：PDF、压缩文件、安装文件等类似的东西。

你可能注意到了，这个清单中没有 HTML。静态的 HTML 页面不算静态资源吗？如果你有，将它们当作静态资源没问题，但那样 URL 要以 .html 结尾，不太“现代化”。尽管可

注 1：借助一些 JavaScript，浏览器可以使用未经编译的 LESS。这种方式会影响性能，所以我不推荐使用。

以创建一个路由，让它以不带后缀名 .html 的方式提供静态 HTML 文件，但一般创建视图（没有任何动态内容的视图）更容易。

注意，如果你只是要搭建 API，可能没有静态内容。这时，你可以跳过这一章。

16.1　性能方面的考虑

如何处理静态资源对网站的性能有很大影响，特别是网站有很多多媒体内容时。在性能上主要考虑两点：*减少请求次数和缩减内容的大小*。

其中减少（HTTP）请求的次数更关键，特别是对移动端来说（通过蜂窝网络发起一次 HTTP 请求的开销要高很多）。有两种办法可以减少请求的次数：合并资源和浏览器缓存。

合并资源主要是架构和前端问题：要尽可能多地将小图片合并到一个子画面中。然后用 CSS 设定偏移量和尺寸只显示图片中需要展示的部分。我强烈推荐用 SpritePad（http://wearekiss.com/spritepad）的免费服务创建子画面。它让子画面的生成容易得不可思议，并且还会生成 CSS。不可能更容易了。SpritePad 的免费功能应该就足够了，但如果你要创建很多子画面，会发现付费也是值得的。

浏览器缓存会在客户端浏览器中存储通用的静态资源，这对减少 HTTP 请求有帮助。尽管浏览器做了很大努力让缓存尽可能自动化，但它也不是完美的：在让浏览器缓存静态资源方面，还有很多你能做也应该做的工作。

最后，我们可以通过缩减静态资源的大小来提升性能。有些技术是*无损的*（不丢失任何数据就可以实现资源大小的缩减），有些技术是*有损的*（通过降低静态资源的品质实现资源大小的缩减）。无损技术包括 JavaScript 和 CSS 的缩小化，以及 PNG 图片的优化。有损技术包括增加 JPEG 和视频的压缩等级。我们会在本章中讨论缩小和打包（也可以减少 HTTP 请求的次数）。

> 在使用 CDN 时一般不用担心 CORS。在 HTML 中加载外部资源不违反 CORS 原则：只有用 AJAX 加载的资源才必须启用 CORS（见第 15 章）。

16.2　面向未来的网站

在将网站放到生产环境中时，静态资源必须放在互联网中的某个地方。你过去可能习惯于把它们放在生成动态 HTML 的服务器上。我们的例子到目前为止用的也是这种方式：输入 `node meadowlark.js` 启动的 Node/Express 服务器会提供所有的 HTML 和静态资源。然而，

如果你想让网站的性能最佳（或者在将来可以这样做），应该希望能轻易地将你的静态资源托管给内容发布网络（CDN）。CDN 是专为提供静态资源而优化的服务器，它利用特殊的头信息（我们马上就会讲到）启用浏览器缓存。另外 CDN 还能基于地理位置进行优化，也就是说它们可以从地理位置上更接近客户端的服务器发布静态内容。尽管互联网确实非常快（虽然不是以光速运行的，但也很接近了），从上百公里的地方发布内容还是比从上千公里的地方快。单次算下来可能节省的时间不多，但如果乘以所有用户、所有请求和所有资源，累加起来就快了。

让网站"面向未来"十分容易，这样当时机到了，你就可以把静态内容挪到 CDN 上，并且我建议你养成这样做的习惯。这归根结底是给静态资源创建一个抽象层，让重新定位它们就像扳动一下开关那么容易。

大部分静态资源都是在 HTML 视图中引用的（指向 CSS 文件的 <link> 元素，指向 JavaScript 文件的 <script>，指向图片的 <img> 标签，以及嵌入多媒体的标签）。然后 CSS 中一般也有静态引用，一般是 background-image 属性。最后，有时在 JavaScript 中也会引用静态资源，比如动态修改或插入 <img> 标签或 background-image 属性。

16.2.1 静态映射

让静态资源可重定位、对缓存友善的策略核心是映射的概念：在编写 HTML 时，我们真的没必要担心静态资源将会放在哪里这种具体细节。我们要关心的是静态资源的逻辑组织。也就是说，重要的是要把 Hood River 度假的照片放在 /img/vacations/hood-river 里，把 Manzanita 的在 /img/vacations/manzanita 里。所以我们的重点是在指定静态资源时让使用这种组织结构变得容易。比如说，在 HTML 中，你希望可以写 <img src="/img/meadowlark_logo.png" alt="Meadowlark Travel Logo">，而不是 <img src="//s3-us-west-2.amazonaws.com/meadowlark/img/meadowlark_logo-3.png" alt="Meadow lark Travel Logo">（如果你用亚马逊的云存储，看起来就是这样的）。

> 我们会用"协议相对 URL"指向静态资源。即 URL 仅以"//"开头，不用"http://"或"https://"。这样浏览器用什么协议都可以。如果用户访问的是安全页面，它会用 HTTPS，否则用 HTTP。很明显，CDN 必须支持 HTTPS，不过我还没见过不支持的。

所以这归根结底是映射的问题：我们要将不太具体的路径（/img/meadowlark_logo.png）映射到更具体的路径（//s3-us-west-2.amazonaws.com/meadowlark/img/meadowlark_logo-3.png）。更进一步讲，我们希望可以随意修改映射。比如，在注册亚马逊 S3 账号之前，你可能希望把图片放在本地（//meadowlarktravel.com/img/meadowlark_logo.png）。

在这些例子中，我们实现映射只是在路径开头添了些东西，我们称之为基准 URL。然而，

你的映射模式可能会更复杂：基本上已经到了极限了。比如说，你可能用一个数据资产数据库将 `''Meadowlark Logo''` 映射到 http://meadowlarktravel.com/img/meadowlark_logo.png。尽管有可能做到，但我警告你不要碰它：使用文件名和路径是相当标准和普遍的内容组织方式，如果要偏离这种方式，你应该有充分的理由。更有实际意义的、更复杂的映射模式案例是采用资产版本化（我们稍后讨论）。比如说，如果草地鹨旅行社的商标经历了 5 次改版，可以写个映射器将 /img/meadowlark_logo.png 映射到 /img/meadowlark_logo-5.png。

眼下我们准备坚持使用非常简单的映射模式：只添加基准 URL。我们假定所有静态资产都是以斜杠开头的。因为映射器要用于几种不同的文件（视图、CSS 和 JavaScript），所以要让它模块化。接下来创建文件 lib/static.js：

```
var baseUrl = '';

exports.map = function(name){
        return baseUrl + name;
}
```

这也没什么，是不是？并且现在它根本什么都没做，只是将参数不加修改地返回了（当然，假定参数是字符串）。没问题，我们现在还是在开发环境中，把静态资源放在本地服务器上挺好。注意，我们可能还要从配置文件中读取 `baseUrl` 的值；现在还是把它留在模块里吧。

> 你可能很想给映射器添个功能，让它检查资产名称是不是以斜杠开头的，如果不是就加一个，但不要忘了，这个资产映射器到处都要用，所以应该尽可能地快。我们可以在 QA 工具链中静态分析代码，确保所有资产名称都是以斜杠开头的。

16.2.2　视图中的静态资源

视图中的静态资源最容易处理，所以先从这里入手。我们可以创建一个 Handlebars 辅助函数（见第 7 章），让它给出一个到静态资源的链接：

```
// 设置 handlebars 视图引擎
var handlebars = require('express3-handlebars').create({
    defaultLayout:'main',
    helpers: {
        static: function(name) {
            return require('./lib/static.js').map(name);
        }
    }
});
```

我们添加了一个 Handlebars 辅助函数 `static`，让它调用静态资源映射器。接下来修改 main.layout，给商标图片用上这个新的辅助函数：

```
<header><img src="{{static '/img/logo.jpg'}}"
    alt="Meadowlark Travel Logo"></header>
```

如果现在运行网站，绝对什么变化都看不到：如果审查源码，会看到商标图片还是 /img/meadowlark_logo.jpg，跟预期的一样。

接下来我们会花些时间把视图和模板中所有对静态资源的引用都改过来。现在 HTML 中的所有静态资源都可以挪到 CDN 上去了。

16.2.3　CSS中的静态资源

CSS 要稍微复杂点，因为没有 Handlebars 帮我们了（配置 Handlebars 以生成 CSS 是有可能的，但它不支持，Handlebars 不是干这个的）。然而像 LESS、Sass 和 Stylus 这样的 CSS 预处理器全都支持变量，这是我们需要的。在这三个流行的预处理器中，我更喜欢 LESS，本书中的例子用的就是 LESS。如果你用的是 Sass 或 Stylus，这些技术很像，并且如何将这项技术用到不同的预处理器上应该很清楚。

我们会给网站添加一个背景图，提供点质感。创建目录 less，并在其中创建文件 main.less：

```
body {
       background-image: url("/img/backgrouind.png");
}
```

这个目前看起来完全是 CSS，并且不是偶然的：LESS 向后兼容 CSS，所以任何有效的 CSS 都是有效的 LESS。事实是，如果在文件 public/css/main.css 中有 CSS，你应该把它们都挪到 less/main.less 中。现在需要一个编译 LESS 生成 CSS 的办法。我们会用 Grunt 任务做这件事：

```
npm install --save-dev grunt-contrib-less
```

然后修改 Gruntfile.js。将 `grunt-contrib-less` 添加到 Grunt 任务列表中加载，然后将下面的代码添加到 `grunt.initConfig` 中：

```
less: {
    development: {
        files: {
            'public/css/main.css': 'less/main.less',
        }
    }
}
```

这段代码基本上读作“从 less/main.less 生成 public/css/main.css”。现在运行 `grunt less`，然后你就会看到 CSS 文件了。现在把它链入布局文件，在 `<head>` 中：

```
        <!-- ... -->
    <link rel="stylesheet" href="{{static /css/main.css}}">
</head>
```

注意，我们用了新做好的 static 辅助函数！这不能解决生成的 CSS 文件中链接到 /img/background.png 的问题，但它确实给 CSS 文件本身创建了可重定位的链接。

现在框架已经搭好了，接下来我们要让 CSS 文件中用的 URL 也是可重定位的。首先我们会将静态映射器作为 LESS 的定制函数。这都可以在 Gruntfile.js 中完成：

```
less: {
    development: {
        options: {
            customFunctions: {
                static: function(lessObject, name) {
                    return 'url("' +
                        require('./lib/static.js').map(name.value) +
                        '")';
                }
            }
        },
        files: {
            'public/css/main.css': 'less/main.less',
        }
    }
}
```

注意，我们给映射器的输出添加了标准的 CSS url 指定器和双引号，这可以确保我们的 CSS 是有效的。现在只需修改 LESS 文件 less/main.less：

```
body {
        background-image: static("/img/background.png");
}
```

注意，真正的改变只是 url 变成了 static。就是这么容易。

16.3　服务器端JavaScript中的静态资源

在服务器端 JavaScript 中使用静态映射器真的很容易，因为我们已经写了一个模块来做映射。比如说，我们想给应用程序添加一个复活节彩蛋。在草地鹨旅行社，我们都是 Bud Clark（前任波特兰市长）的狂热粉丝。所以我们想在 Clark 生日那天把商标换成他的照片。修改 meadowlark.js：

```
var static = require('./lib/static.js').map;

app.use(function(req, res, next){
        var now = new Date();
        res.locals.logoImage = now.getMonth()==11 && now.getDate()==19 ?
                static('/img/logo_bud_clark.png') :
                static('/img/logo.png');
        next();
});
```

然后在 views/layouts/main.handlebars 中：

```
<header><img src="{{logoImage}}" alt="Meadowlark Travel Logo"></header>
```

注意，我们在视图中没用 Handlebars 的 `static` 辅助函数：因为已经在路由处理器里用了，如果这里再用，就是映射了两次，那就糟糕了！

16.4 客户端JavaScript中的静态资源

你可能直觉地认为将静态映射放到客户端很简单，并且对于我们这种简单的情况，它完全可以胜任（尽管必须用 browserify 才能在浏览器中使用 Node 风格的模块）。然而我要给你泼冷水了，因为随着映射器变得越来越复杂，它很快就会崩溃。比如说，如果我们开始用数据库实现更复杂的映射，在浏览器中就不能再用了。然后我们可能必须用上 AJAX 调用，那样服务器才能为我们映射文件，这会很大程度上减慢速度。

那怎么办呢？好在这里有个简单的解决方案。虽然不像访问映射器那么优雅，但它完全不会为我们带来其他问题。

比如说，你用 jQuery 动态修改购物车的图片：当它是空的时，视觉效果上是一个空的购物车。当用户往里面添了东西后，购物车里会出现一个盒子。（我们真的想用子画面实现这个功能，但为了演示这个例子会用两张不同的图片。）

这两张图片是 /img/shop/cart_empty.png 和 /img/shop/cart_full.png。没有映射，我们大概会这样做：

```
$(document).on('meadowlark_cart_changed'){
        $('header img.cartIcon').attr('src', cart.isEmpty() ?
                '/img/shop/cart_empty.png' : '/img/shop/cart_full.png' );
}
```

在我们将图片挪到 CDN 上后，这就不行了，所以我们也想映射这些图片。解决方案是在服务器端映射，然后设定定制的 JavaScript 变量。在 views/layouts/main.handlebars 中这样做：

```
<!-- ... -->
<script>
    var IMG_CART_EMPTY = '{{static '/img/shop/cart_empty.png'}}';
    var IMG_CART_FULL = '{{static '/img/shop/cart_full.png'}}';
</script>
```

然后只要在 jQuery 中使用那些变量：

```
$(document).on('meadowlark_cart_changed', function(){
        $('header img.cartIcon').attr('src', cart.isEmpty() ?
                IMG_CART_EMPTY : IMG_CART_FULL );
});
```

如果你要在客户端做很多的图片切换，可能要把所有图片变量放在一个对象中（它本身就成了一个映射）。比如可以这样重写前面的代码：

```
<!-- ... -->
<script>
        var static = {
                IMG_CART_EMPTY: '{{static '/img/shop/cart_empty.png'}}',
                IMG_CART_FULL: '{{static '/img/shop/cart_full.png'}}
        }
</script>
```

16.5　提供静态资源

现在我们已经明白如何创建一个框架来轻松地修改静态资源的提供源了，那么究竟什么才是存储这些资产的最佳方式呢？了解浏览器用来确定如何（以及是否）缓存的响应头会有帮助。

- `Expires/Cache-Control`

  这两个响应头信息告诉浏览器一个资源可以缓存的最长时间。浏览器会认真对待它们：如果它们告诉浏览器某个资源要缓存一个月，那么在这一个月里只要缓存中有这个资源，浏览器绝不会重新下载。一定要知道，出于某些你不可控的原因，浏览器可能会提前从缓存中移除图片。比如用户手动清除了缓存，或浏览器为了给用户访问更频繁的某些资源腾出空间清除了你的资源。你只需要其中一个响应头，支持 `Expires` 的更多，所以应该优先选择它。如果资源在缓存中，而且它还没过期，浏览器就绝对不会发起 `GET` 请求，这会提升性能，特别是在移动端上。

- `Last-Modified/ETag`

  这两个标签提供了某种版本化：如果浏览器需要获取资源，它会在下载之前检查这些标签。还会向服务器发起 `GET` 请求，但如果这些响应头返回的值让浏览器觉得资源没变，它就不会继续下载那个文件。如其名所示，`Last-Modified` 可以指定资源最后一次修改的时间。`ETag` 可以是任意字符串，一般是版本字符串或内容的哈希值。

在提供静态资源时，你应该用 `Expires` 响应头，加上 `Last-Modified` 或 `ETag`。Express 内置的静态中间件会设定 `Cache-Control`，但不会处理 `Last-Modified` 或 `ETag`。所以只适合在开发环境中使用，对于生产环境来说不是太好。

如果你选择把静态资源放在 CDN 上，比如亚马逊 CloudFront、微软 Azure 或 MaxCDN，好处是它们会帮你处理好大部分细节。你可以对这些细节进行微调，但这些服务提供的默认值已经很好了。

如果你不想把静态资源放到 CDN 上，但想要比 Express 内置的 `connect` 中间件更健壮的方案，可以考虑用代理服务器，比如 Nginx（见第 12 章），它完全可以胜任。

16.6 修改静态内容

缓存极大提升了网站的性能，但也不是没有代价的。特别是如果你修改了静态资源，客户可能直到浏览器中缓存的版本过期后才能见到。谷歌推荐缓存一个月，最好是一年。想象一个每天在相同浏览器上使用网站的用户：那个人可能一整年都没看到你的更新！

很明显我们不想出现这种局面，但你也不能告诉用户让他们清除缓存。解决方案是*指纹法*。指纹法只是在资源名上加上某种版本信息。你更新资产后，资源名称会变，浏览器就知道它需要下载这个资源了。

以商标为例（/img/meadowlark_logo.png）。如果为了性能最佳把它放在了 CDN 上，指定有效期为一年，然后我们修改了它，用户可能一年后才能见到更新后的商标。然而，如果将商标重命名为 /img/meadowlark_logo-1.png（并把名称的变化反映到 HTML 中），浏览器就会被强制下载它，因为它看起来是新资源。

如果你的网站上有几十、上百甚至上千张图片，这种方式看起来可能非常笨重。如果是这样（有大量放在 CDN 上的图片），你可能想把静态映射器做得更精巧。比如说，你可能在数据库中保存所有数据资产的当前版本，然后静态映射器可以查找资产名称（比如 /img/meadowlark_logo.png），然后返回那个资产最新版的 URL（/img/meadowlark_logo-12.png）。

最起码应该给 CSS 和 JavaScript 文件加上指纹。商标不是最新的是一码事，但如果推出一个新功能，或者修改了页面布局，然后发现因为资源被缓存了用户看不到，那就太烦人了。

除了单个文件的指纹，另外一个流行的方案是资源打包。打包即把所有 CSS 捣烂到一个人类几乎不可能看懂的文件中，客户端 JavaScript 也是如此。既然总会创建新文件，一般做那些文件的指纹更容易也更常见。

16.7 打包和缩小

在减少 HTTP 请求次数和缩减通过网络发送数据的努力中，“打包和缩小”流行了起来。打包将多个文件（CSS 或 JavaScript）打到一个文件中（从而减少 HTTP 请求次数）。缩小将源码中所有不必要的东西都去掉，比如空格（字符串外面的），它甚至可以将变量名变得更短。

打包和缩小还有一个额外的好处，即减少了需要做指纹处理的资产数量。事情仍然会很快变得复杂起来！好在有些 Grunt 任务能帮我们控制这种疯狂的局面。

因为我们的项目目前还没有客户端 JavaScript，所以我们先创建两个文件：一个用于“联系我们”的表单提交处理，另一个用于购物车功能。我们现在只是在里面放一些日志，以

便可以验证打包和缩小可用：

public/js/contact.js:

```
$(document).ready(function(){
    console.log('contact forms initialized');
});
```

public/js/cart.js:

```
$(document).ready(function(){
    console.log('shopping cart initialized');
});
```

我们已经有了一个 CSS 文件（从 LESS 文件生成的），但还是再添加一个。我们把与购物车相关的样式放到它们自己的 CSS 文件 less/cart.less 中：

```
div.cart {
    border: solid 1px black;
}
```

现在在 Gruntfile.js 中把它添加的 LESS 文件列表中编译：

```
files: {
    'public/css/main.css': 'less/main.less',
    'public/css/cart.css': 'less/cart.css',
}
```

为了达成目标，我们至少要用到 3 个 Grunt 任务：一个用于 JavaScript，一个用于 CSS，另外一个用来做文件的指纹。接下来我们先安装这些模块：

```
npm install --save-dev grunt-contrib-uglify
npm install --save-dev grunt-contrib-cssmin
npm install --save-dev grunt-hashres
```

然后在 Gruntfile 中加载这些任务：

```
[
    // ...
    'grunt-contrib-less',
    'grunt-contrib-uglify',
    'grunt-contrib-cssmin',
    'grunt-hashres',
].forEach(function(task){
    grunt.loadNpmTasks(task);
});
```

并设置这些任务：

```
    grunt.initConfig({
        // ...
```

```
        uglify: {
            all: {
                files: {
                    'public/js/meadowlark.min.js': ['public/js/**/*.js']
                }
            }
        },
        cssmin: {
            combine: {
                files: {
                    'public/css/meadowlark.css': ['public/css/**/*.css',
                       '!public/css/meadowlark*.css']
                }
            },
            minify: {
                src: 'public/css/meadowlark.css',
                dest: 'public/css/meadowlark.min.css',
            }
        },
        hashres: {
            options: {
                fileNameFormat: '${name}.${hash}.${ext}'
            },
            all: {
                src: [
                    'public/js/meadowlark.min.js',
                    'public/css/meadowlark.min.css',
                ],
                dest: [
                    'views/layouts/main.handlebars',
                ]
            },
        }
    });
};
```

我们来看看刚才做了什么。在 `uglify` 任务中（缩小经常被称为“丑化”是因为……好吧，只要看一下输出，你就明白了），我们把网址的所有 JavaScript 拿到一起放到一个文件 meadowlark.min.js 中。对于 `cssmin` 而言，我们有两个任务：首先把所有 CSS 放到一个 meadowlark.css 文件中（注意那个数组中的第二个元素：字符串前面那个感叹号说不要包含那些文件……这样可以防止它循环包含它自己生成的文件！）。然后我们缩小合并的 CSS 到 meadowlark.min.css 文件中。

在讲 `hashres` 之前，我们先暂停一秒。我们已经把所有 JavaScript 放进了 meadowlark.min.js，所有 CSS 放进了 meadowlark.min.css。

现在，我们不要在 HTML 中引用单个文件，而是要在布局文件中引用它们。所以接下来要修改布局文件：

```
    <!-- ... -->
    <script src="http://code.jquery.com/jquery-2.0.2.min.js"></script>
    <script src="{{static '/js/meadowlark.min.js'}}"></script>
    <link rel="stylesheet" href="{{static '/css/meadowlark.min.css'}}">
</head>
```

到目前为止，看起来像是为了很小的回报做了很多工作。然而随着网站的增长，你会发现自己添加了越来越多的 JavaScript 和 CSS。我见过有几十或更多 JavaScript 文件以及五六个 CSS 文件的项目。一旦达到这种数量，打包和缩小会产生极大的性能提升。

现在讲 hashres 任务。我们想给这些打包和缩小的 CSS 和 JavaScript 文件添加指纹，以便在更新网站时可以马上看到这些变化，而不是要等到缓存的版本到期。hashres 任务帮我们处理这种复杂性。注意，我们告诉它想要重命名 public/js/meadowlark.min.js 和 public/css/meadowlark.min.css 文件。hashres 会生成文件的哈希（一个数学指纹）并追加到文件名上。所以现在不再是 /js/meadowlark.min.js，而是 /js/meadowlark.min.62a6f623.js（你的版本只要有一个字符不同，实际的哈希值就会不一样）。如果你每次都要记住修改 views/layout/main.handlebars 中的引用，好吧……你有时可能会忘掉。好在 hashres 任务可以解救你，它可以自动修改那些引用。我们在配置中是如何在 dest 部分指定 views/layouts/main.handlebars 的？那会自动帮我们修改引用。

那么现在试一下吧。按正确的顺序做事情很重要，因为这些任务有依赖关系：

```
grunt less
grunt cssmin
grunt uglify
grunt hashres
```

每次修改 CSS 和 JavaScript 都有很多工作要做，所以我们要设置一个 Grunt 任务，这样就不用记住这些了。修改 Gruntfile.js：

```
grunt.registerTask('default', ['cafemocha', 'jshint', 'exec']);
grunt.registerTask('static', ['less', 'cssmin', 'uglify', 'hashres']);
```

现在我们只要输入 grunt static，一切事情就都被做好了。

在开发模式中跳过打包和缩小

打包和缩小有个问题，即用了之后不可能做前端调试了。所有 JavaScript 和 CSS 都被捣碎在它们自己的包中，如果你选择了非常积极的缩小选项，情况就会更糟。理想的做法是在开发模式中禁用打包和缩小。好在我为此写了个模块：connect-bundle。

在用这个模块之前，我们先创建一个配置文件。我们现在定义打包，但稍后还要用这个配置文件指定数据库配置。一般配置文件会用 JSON 格式，并且有一个少有人知但非常实用的技巧，可以用 require 读取和解析 JSON 文件，就好像它是个模块一样：

```
var config = require('./config.json');
```

然而因为我厌烦了输入引号，所以一般更愿意把配置放在 JavaScript 文件中（几乎跟 JSON 文件一样，只是少了几个引号）。接下来创建 config.js：

```
module.exports = {
    bundles: {

        clientJavaScript: {
            main: {
                file: '/js/meadowlark.min.js',
                location: 'head',
                contents: [
                    '/js/contact.js',
                    '/js/cart.js',
                ]
            }
        },

        clientCss: {
            main: {
                file: '/css/meadowlark.min.css',
                contents: [
                    '/css/main.css',
                    '/css/cart.css',
                ]
            }
        }
    }
}
```

我们定义了 JavaScript 和 CSS 的打包。打包可以有多个（比如一个用于桌面端，一个用于移动端），但我们的例子只有一个，称为 main。注意，在 JavaScript 打包中，我们可以指定位置。出于性能和依赖方面的原因，你可能会把 JavaScript 放在不同的位置。在 <head> 中，紧跟在 <body> 开始标签后面，或者就在 </body> 结束标签前面，这些都是常见的引入 JavaScript 的位置。这里我们指定了 head（可以随意叫它什么，但 JavaScript 打包必须有个位置）。接下来修改 views/layouts/main.handlebars：

```
    <!-- ... -->
    {{#each _bundles.css}}
        <link rel="stylesheet" href="{{static .}}">
    {{/each}}
    {{#each _bundles.js.head}}
        <script src="{{static .}}"></script>
    {{/each}}
</head>
```

现在如果我们想用指纹化的打包名，必须修改 config.js，而不是 views/layouts/main.handlebars。还要相应地修改 Gruntfile.js：

```
hashres: {
    options: {
        fileNameFormat: '${name}.${hash}.${ext}'
    },
    all: {
        src: [
            'public/js/meadowlark.min.js',
            'public/css/meadowlark.min.css',
        ],
        dest: [
            'config.js',
        ]
    },
}
```

现在运行 `grunt static`，你会看到 config.js 中的打包名的指纹已经被更新了。

16.8　关于第三方库

你应该注意到了，这些例子中所有打包里都没有 jQuery。jQuery 如此普遍，我觉得把它放在包里没什么价值：很可能浏览器已经缓存了。像 Handlebars、Backbone 或 Bootstrap 这些库可能是灰色区域：它们十分流行，但还没达到浏览器缓存中一直会有的地步。如果你只用一两个第三方库，可能没必要把它们和你的脚本一起打包。不过如果你有五个或者更多库，可能会见到打包这些库后性能上的提升。

16.9　QA

与其等着不可避免的 bug 出现，或者希望代码审查能抓住问题，何不在我们的 QA 工具链中添加个组件解决问题呢？我们将会用到一个 Grunt 插件 `grunt-lint-pattern`，它只是在源码文件中搜索特定的模式，发现后就生成一个错误。先安装这个包：

```
npm install --save-dev grunt-lint-pattern
```

然后将 `grunt-lint-patter` 添加到 Gruntfile.js 要加载的模块列表中，然后添加下面的配置：

```
lint_pattern: {
    view_statics: {
        options: {
            rules: [
                {
                   pattern: /<link [^>]*href=["'](?!\{\{static )/,
                   message: 'Un-mapped static resource found in <link>.'
                },
                {
                   pattern: /<script [^>]*src=["'](?!\{\{static )/,
                   message: 'Un-mapped static resource found in <script>.'
                },
```

```
                {
                    pattern: /<img [^>]*src=["'](?!\{\{static )/,
                    message: 'Un-mapped static resource found in <img>.'
                },
            ]
        },
        files: {
            src: [
                'views/**/*.handlebars'
            ]
        }
    },
    css_statics: {
        options: {
            rules: [
                {
                    pattern: /url\(/,
                    message: 'Un-mapped static found in LESS property.'
                },
            ]
        },
        files: {
            src: [
                'less/**/*.less'
            ]
        }
    }
}
```

并将 `lint_pattern` 添加到默认规则中：

```
grunt.registerTask('default', ['cafemocha', 'jshint', 'exec', 'lint_pattern']);
```

现在运行 `grunt` 时（我们应该定期这样做），会抓到所有未映射的静态实例。

16.10 小结

对于看起来这么简单的一件事，静态资源的麻烦够多的。然而，它们可能代表着要真正传输给访问者的大量数据，所以花点时间优化会产生可观的回报。

对某些不太大也不复杂的网站来说，我在这里罗列出来的静态资源映射技术可能有点威力过大了。对于那样的项目，另一个可行的方案是一开始就把静态资源放在 CDN 上，并一直在视图和 CSS 中使用完整的 URL。你可能仍想要运行某种程序分析工具（linting）来确保没把静态资源放在本地：可以用 `grunt-lint-pattern` 搜索不以 `(?:https?:)?//;` 开头的链接，防止你不小心用了本地资源。

如果你的应用程序不值得为静态资源做这么多工作，精减打包和缩小也可以帮你节省时间。特别是网站只有一两个 JavaScript 文件时，并且所有 CSS 都放在一个文件里，你可能

也可以跳过打包；除非 JavaScript 或 CSS 很大，否则缩小得到的收获也不大。

不管选择什么技术提供静态资源，我都会强烈建议把它们单独部署，并优选 CDN。如果你觉得麻烦，我可以向你保证它并不像听上去那么困难，特别是你在部署系统上花点时间后，把静态资源部署到一个地方，程序部署到另一个地方，整个过程都是自动的。

如果你担心 CDN 的费用问题，我建议你去看一下现在部署所花的费用。大多数托管服务提供商基本上都是在对带宽收费，即便你不知道。然而，如果忽然有一天网站被 Slashdot 提到了，你被“Slashdotted”了，你可能会被托管账单吓到。CDN 托管通常会被设置成只为使用的东西付费。举个例子，我为一家中等规模的区域性公司管理的网站（并且是一个多媒体资源丰富的网站），一个月用 20GB 的带宽，每个月只为托管静态资源付几美元。

把静态资源托管在 CDN 上所能得到的性能提升十分显著，并且这样做的成本和不便之处微乎其微，所以我强烈建议你这样做。

第 17 章

在Express中实现MVC

我们已经介绍过很多基础知识了，如果你觉得已经有点儿吃不消了，别担心，大家都有这种感觉。这一章我们要讨论的技术能让这种疯狂的局面有点儿秩序。

最近这些年流行起来的开发范式中比较突出的一个就是模型 – 视图 – 控制器（MVC）模式。这个概念相当古老了，实际上可以追溯到 20 世纪 70 年代。它的复兴要归功于它在 Web 开发领域中的适用性。

据我观察，MVC 最大的优势之一是它减少了项目的学习时间。比如说，一个熟悉 MVC 框架的 PHP 开发人员可以非常轻松地进入一个 .NET MVC 项目。实际上编程语言一般并不能形成什么障碍，只要知道到哪里找东西就行。MVC 将功能分解到有明确定义的领域中，给了我们一个通用的软件开发框架。

在 MVC 中，模型是“纯粹”的数据和逻辑。它根本不关心自己跟用户之间的交互。视图将模型传递给用户，而控制器则接受用户输入，处理模型，选择要显示哪个（些）视图。（我经常想，“调度器”应该比“控制器”更合适：毕竟，控制器听起来不像是会接受用户输入的东西，而在 MVC 项目中这是控制器的一个主要责任。）

MVC 已经繁衍出了数不清的变体。微软的“模型 – 视图 – 视图模型”（MVVM）特别引入了一个重要概念：视图模型（它把控制器也推到了视图中，我觉得是个没什么意思的简化）。视图模型的想法是说它是模型的转化。另外，单个视图模型可能由不止一个模型组成，或者是几个模型的部分，或者是单个模型的部分。乍一看可能觉得没必要搞那么复杂，但我发现这个概念很有价值。它的价值在于可以“保护”模型。在纯粹的 MVC 中，它会引诱（甚至是强迫）你对模型做只对视图来说有必要的转换或改进。模型视图可以

“解救”你：如果你需要一个只用来展示的数据视图，它就属于视图模型。

跟其他任何模式一样，你必须决定在贯彻这一模式时要多严格。过于严格会导致为了边缘情况的“正确做法”做出英勇就义式的努力，而过于松散又会导致维护问题和技术债务问题。我倾向于尽可能地向严格那一面倾斜。幸好，MVC（及视图模型）提供了非常自然的职责区域，并且我发现极少能碰到这个模式无法轻松容纳的情况。

17.1 模型

对我来说，模型绝对是最最重要的组件。如果你的模型足够健壮并设计优良，你总能废掉表示层（或者添加一个额外的表示层）。可换个方向就困难多了：模型是项目的基石。

千万不要用表示代码或用户交互代码污染了你的模型。即便它看起来更容易或更有利，我可以向你保证，你只是在给自己挖坑。一个更复杂也更有争议的问题是模型和持久层之间的关系。

在理想情况下，模型和持久层是可以完全分开的。这肯定是可以达到的，但通常需要付出很大的代价。非常普遍的情况是，模型中的逻辑严重依赖于持久性，把这两层分开可能得不偿失。

本书所采取的是阻力最小的路线，用 Mongoose（专门针对 MongoDB 的）来定义模型。如果绑定到特定的持久化技术上让你觉得不安，你可能要考虑使用原生的 MongoDB 驱动（不需要任何方案或对象映射），并把你的模型跟持久层分开。

有人提出模型应该仅包含数据。也就是说没有逻辑，只有数据。尽管“模型”这个词使人更多的想到数据而不是功能，但我发现这个限制没什么用处，所以更喜欢将模型看作数据和逻辑的结合体。

我建议你在项目中创建一个叫 models 的子目录来存放模型。只要你有要实现的逻辑，或要存储的数据，都应该在 models 目录下的文件里完成。比如说，你可能要把客户数据和逻辑放在文件 models/customer.js 中：

```
var mongoose = require('mongoose');
var Orders = require('./orders.js');

var customerSchema = mongoose.Schema({
        firstName: String,
        lastName: String,
        email: String,
        address1: String,
        address2: String,
        city: String,
        state: String,
        zip: String,
```

```
		phone: String,
		salesNotes: [{
			date: Date,
			salespersonId: Number,
			notes: String,
		}],
});

customerSchema.methods.getOrders = function(){
		return Orders.find({ customerId: this._id });
};

var Customer = mongoose.model('Customer', customerSchema);
modules.export = Customer;
```

17.2 视图模型

尽管我不想在面对“将模型直接传递给视图”这种问题时表现得太教条，但如果你只是因为要在视图中显示什么就忍不住要修改你的模型，那我肯定会建议你创建个视图模型。视图模型是保持模型抽象性的办法，同时还能为视图提供有意义的数据。

还用前面那个例子。我们有个 `Customer` 模型。现在要创建一个视图显示客户信息，还有一串订单。然而我们的 `Customer` 模型不太好用。里面有我们不想显示的数据（销售记录），并且我们要格式化不同的数据（比如正确格式化邮件地址和电话号码）。更进一步说，我们想要显示不在 `Customer` 模型中的数据，比如客户订单列表。这时用视图模型就会很方便。接下来我们在 viewModels/customer.js 中创建一个视图模型：

```
var Customer = require('../model/customer.js');

// 联合各域的辅助函数
function smartJoin(arr, separator){
		if(!separator) separator = ' ';
		return arr.filter(function(elt){
			return elt!==undefined &&
				elt!==null &&
				elt.toString().trim() !== '';
		}).join(separator);
}

module.exports = function(customerId){
		var customer = Customer.findById(customerId);
		if(!customer) return {	error: 'Unknown customer ID: ' +
			req.params.customerId };
		var orders = customer.getOrders().map(function(order){
			return {
				orderNumber: order.orderNumber,
				date: order.date,
				status: order.status,
				url: '/orders/' + order.orderNumber,
```

```
                }
        });
        return {
                firstName: customer.firstName,
                lastName: customer.lastName,
                name: smartJoin([customer.firstName, customer.lastName]),
                email: customer.email,
                address1: customer.address1,
                address2: customer.address2,
                city: customer.city,
                state: customer.state,
                zip: customer.zip,
                fullAddress: smartJoin([
                        customer.address1,
                        customer.address2,
                        customer.city + ', ' +
                                customer.state + ' ' +
                                customer.zip,
                ], '<br>'),
                phone: customer.phone,
                orders: customer.getOrders().map(function(order){
                        return {
                                orderNumber: order.orderNumber,
                                date: order.date,
                                status: order.status,
                                url: '/orders/' + order.orderNumber,
                        }
                }),
        }
}
```

在这个代码示例中，你能看到我们如何丢掉不需要的信息，如何重新格式化一些信息（比如 `fullAddress`），甚至如何构造额外的信息（比如用来获取订单详情的 URL）。

视图模型的概念对于保护模型的完整性和范围是必不可少的。如果你需要所有的副本（比如 `firstname: customer.firstName`），你可能想要看看 Underscore（http://underscorejs.org/），用它可以做更多精心的对象组成。比如说，你可以克隆一个对象，只挑选你想要的属性，或者相反，克隆对象时忽略特定的属性。下面用 Underscore 重写了上一个例子（install with `npm install --save underscore`）：

```
var _ = require('underscore');
// 得到一个客户视图模型
function getCustomerViewModel(customerId) {
        var customer = Customer.findById(customerId);
        if(!customer) return { error: 'Unknown customer ID: ' +
                req.params.customerId };
        var orders = customer.getOrders().map(function(order){
                return {
                        orderNumber: order.orderNumber,
                        date: order.date,
                        status: order.status,
```

```
                        url: '/orders/' + order.orderNumber,
                }
        });
        var vm = _.omit(customer, 'salesNotes');
        return _.extend(vm, {
                name: smartJoin([vm.firstName, vm.lastName]),
                fullAddress: smartJoin([
                        customer.address1,
                        customer.address2,
                        customer.city + ', ' +
                                customer.state + ' ' +
                                customer.zip,
                ], '<br>'),
                orders: customer.getOrders().map(function(order){
                        return {
                                orderNumber: order.orderNumber,
                                date: order.date,
                                status: order.status,
                                url: '/orders/' + order.orderNumber,
                        }
                }),
        });
}
```

注意，我们还用了 JavaScript 的 .map 方法给客户视图模型设定订单列表。从本质上来讲，我们创建了一个临时（或匿名）的视图模型。还有一种方式是创建一个“客户订单视图模型”对象。如果我们要在多处使用那个视图模型，第二种方式更好。

17.3　控制器

控制器负责处理用户交互，并根据用户交互选择恰当的视图来显示。听起来是不是很像请求路由？实际上，控制器和路由器之间唯一的区别是控制器一般会把相关功能归组。我们已经见过一些把相关路由分组的办法了，现在只是通过管它叫控制器来做得更正式。

想象有一个“客户控制器”，它负责客户信息的显示和编辑，包括客户下的订单。我们来创建一个这样的控制器，controllers/customer.js：

```
var Customer = require('../models/customer.js');
var customerViewModel = require('../viewModels/customer.js');

exports = {

        registerRoutes: function(app) {
                app.get('/customer/:id', this.home);
                app.get('/customer/:id/preferences', this.preferences);
                app.get('/orders/:id', this.orders);
                app.post('/customer/:id/update', this.ajaxUpdate);
        }

        home: function(req, res, next) {
```

```
        var customer = Customer.findById(req.params.id);
        if(!customer) return next(); // 将这个传给 404 处理器
        res.render('customer/home', customerViewModel(customer));
    }

    preferences: function(req, res, next) {
        var customer = Customer.findById(req.params.id);
        if(!customer) return next(); // 将这个传给 404 处理器
        res.render('customer/preferences', customerViewModel(customer));
    }

    orders: function(req, res, next) {
        var customer = Customer.findById(req.params.id);
        if(!customer) return next(); //  将这个传给 404 处理器
        res.render('customer/preferences', customerViewModel(customer));
    }

    ajaxUpdate: function(req, res) {
        var customer = Customer.findById(req.params.id);
        if(!customer) return res.json({ error: 'Invalid ID.'});
        if(req.body.firstName){
            if(typeof req.body.firstName !== 'string' ||
                req.body.firstName.trim() === '')
                return res.json({ error: 'Invalid name.'});
            customer.firstName = req.body.firstName;
        }
        // 等等……
        customer.save();
        return res.json({ success: true });
    }
}
```

注意，在这个控制器中，我们将路由管理跟真正的功能分开了。在这个例子里，home、preferences 和 orders 方法除了所选的视图不同，其他都是一样的。如果我们做的只是这些，我可能会把它们合到一个通用的方法中，但在这里做成这样是因为它们可能会被进一步定制化。

这个控制器中最复杂的方法是 ajaxUpdate。从名字就能看出来，我们会在前端用 AJAX 做更新。要注意的是，我们没有盲目地根据请求体中传来的参数更新客户对象，那样我们可能会遭受攻击。单个处理各域要做更多工作，但更安全。还有，我们要进行校验，即便我们在前端也做了。记住，攻击者可能会检查你的 JavaScript，并构造一个 AJAX 查询绕过你的前端校验，试图欺骗你的程序，所以即便是冗余的，也一定要在服务器端做校验。

能限制你的选择的只有你的想象力了。如果你想把控制器从路由中完全剥离出来，你肯定可以那么做。按我的观点，那种抽象是没必要的，但如果你要尝试写一个可以处理不同 UI（比如本地程序）的控制器，那样做可能是有意义的。

17.4　小结

像很多编程范式或模式一样，MVC 是一个比特定技术更通用的概念。你在这一章里已经见过了，我们所采取的办法几乎都在这里了：我们只是给路由处理器起了个“控制器”的名字，并把路由跟功能分开了，这样显得更加正式一点儿。我们还介绍了视图模型的概念，我觉得它对保持模型的整体性至关重要。

第 18 章 安全

现在大多数网站和应用程序都会有某种安全方面的需求。如果你允许人们登录，或者储存了个人身份信息（PII），就要给网站实现某种安全机制。

本章会讨论 HTTPS（HTTP Secure，这是建立安全网站的基础），以及认证机制，并重点讨论第三方认证。

安全本身就是能写一本书的大课题。因此，本书的重点是利用已有的认证模块。自己编写认证系统肯定是可能的，但那是一个大型而复杂的任务。另外，选择第三方登录有很好的理由，这点我们会在后面讨论。

18.1 HTTPS

使用 HTTPS 是提供安全服务的第一步。互联网的本质决定了第三方有可能截取客户端和服务器端之间传输的数据包。HTTPS 会对那些包进行加密，让攻击者极难访问到所传输的信息。（我是说非常困难，不是不可能，因为没有完美的安全这种东西。然而，业内认为 HTTPS 对银行、企业安全和医疗保健都是足够安全的。）

你可以把 HTTPS 当作确保网站安全的基础。它不提供认证，但为认证奠定了基础。比如说，认证系统可能涉及传输密码：如果密码是未经加密进行传输的，再复杂的认证也不能确保系统的安全。安全的强度取决于整个体系中最弱的一环，而其中第一环就是网络协议。

HTTPS 协议基于服务器上的*公钥证书*，有时也叫 SSL 证书。SSL 证书目前的标准格式是 X.509。证书背后的思想是由证书颁发机构（CA）发行证书。CA 让浏览器厂商能访问受

信根证书。在你安装浏览器时，其中就包含这些受信根证书，并靠它们建立起 CA 和浏览器之间的信任链。要用这个信任链，你的服务器必须使用由 CA 颁发的证书。

结果是要提供 HTTPS，则需要有来自 CA 的证书，那怎么才能得到这样的证书呢？大体上有三种途径：你可以自己生成，也可以从免费 CA 那里获取，或者从商业 CA 那里买一个。

18.1.1　生成自己的证书

生成证书很容易，但一般只适用于开发和测试用途（还有可能是部署在内网中）。由于 CA 确立起来的层级性，浏览器只信任由已知 CA（并且那个可能不是你）生成的证书。如果你的证书来自浏览器不知道的 CA，浏览器会用非常惊悚的语言警告你，说你正在用一个未知（因此是不可信的）实体建立安全连接。这在开发和测试过程中没什么问题：你和你的团队知道你们是自己生成的证书，并且你知道浏览器会这样。如果你把这样一个网站部署到生产环境中让公众访问，他们会成群结队地离开的。

> 如果你们能控制浏览器的发行和安装，可以在安装浏览器时自动装上你们自己的根证书，这样人们在连接网站时就不会看到警告信息了。然而要做到这一点并不容易，仅适用于由你们控制用哪个浏览器的环境中。除非你有非常充分的理由，否则这样做一般是得不偿失的。

要生成自己的证书，你需要一个 OpenSSL 实现。表 18-1 给出了获取这样一个实现的方法。

表18-1　在不同的平台上获取OpenSSL实现

平台	指令
OS X	`brew install openssl`
Ubuntu、Debian	`sudo apt-get install openssl`
Other Linux	从 http://www.openssl.org/source/ 下载，解压压缩包并按指示操作
Windows	从 http://gnuwin32.sourceforge.net/packages/openssl.htm 下载

> 如果你是 Windows 用户，可能需要指定 OpenSSL 配置文件的位置，因为 Windows 路径名称的问题，这需要些技巧。万无一失的办法是定位到 openssl.cnf 文件（通常在安装的共享目录中），并在运行 `openssl` 命令之前设定环境变量 `OPENSSL_CNF`：`SET OPENSSL_CONF=openssl.cnf`。

装上 OpenSSL 之后就可以生成私钥和公共证书：

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout meadowlark.pem
      -out meadowlark.crt
```

它会问你一些细节信息，比如国家代码、城市、省（州）、全限定域名（FQDN）、邮件地

址等。既然这个证书是用于开发 / 测试的，你怎么回答关系不大（实际上它们都是可选的，但不回答会让浏览器觉得这个证书更可疑）。通用名（FQDN）是浏览器用来识别域名的。所以如果你用的是本地服务器，可以用它做 FQDN，或者用服务器的 IP 地址，或者服务器名，如果可用的话。即便通用名和你用在 URL 中的域名不一致，也仍然可以加密，但浏览器会就此差异给你一个额外的警告。

如果你对这条命令的细节感到好奇，可以看看 OpenSSL 文档（http://www.openssl.org/docs/apps/req.html）。这里要指出的是，选项 `-nodes` 跟 Node 没任何关系，甚至也不是复数“nodes”，它真正的意思是“no DES”，表示私钥不是 DES 加密的。

这个命令会生成两个文件：meadowlark.pem 和 meadowlark.crt。PEM（Privacy-enhanced Electronic Mail，保密增强电子邮件）文件是你的私钥，不应该让客户端访问到。CRT 文件是自签名证书，会发送给浏览器建立安全链接。

此外还有提供免费自签名证书的网站，比如 http://www.selfsignedcertificate.com。

18.1.2　使用免费的证书颁发机构

HTTPS 是基于信任的，而在互联网上获得信任最简单的方式是购买它，这是个令人不爽的现实。并且它也不是万金油：建立安全基础设施、投保证书，以及维护跟浏览器厂商之间的关系都很昂贵。然而你并不是只能购买用于生产环境中的证书：CACert（http://www.cacert.org）采用基于积分的“信任网”来确保你身份的真实性。要获取足够的积分来获得证书，你必须会见一位 CACert 成员，即有资质的“担保人”。或者你也可以出席能获得积分的活动。

可惜，一分钱一分货：目前没有主流浏览器支持 CACert。可能它最终会得到 Mozilla Firefox 的支持，但考虑到 CACert 的非盈利性质，它基本不可能得到谷歌 Chrome、IE 或苹果 Safari 的支持。

因此我只能建议你只在开发和测试中使用 CACert 证书，或者你的服务是专门针对开源人群的，他们不会被不可信的证书吓倒。

所有主流的证书厂商（比如 Comodo 和 Symantec）都提供 30 天到 90 天的免费试用证书。如果你要测试商用证书，这是个有效的选择，但如果你要保证服务的持续性，则要在试用期结束之前购买证书。

18.1.3　购买证书

现在跟着每个主流浏览器发行的 50 个根证书中有将近 90% 是属于四家公司的：Symantec（收购了 VeriSign）、Comodo 集团、Go Daddy 和 GlobalSign。直接从 CA 购买可能十分昂

贵：一般是每年 300 美元起（尽管有些收费每年不到 100 美元）。通过代理商购买会便宜些，能买到每年 10 美元或更便宜的 SSL 证书。

知道你在花钱买什么，以及你为什么要为一个证书支付 10 美元、150 美元或 300 美元（甚至更多）很重要。首先你要搞清楚，10 美元的证书跟 1500 美元的证书在加密级别上没有任何差别。给出高价格的 CA 希望你最好不知道这一点：他们的销售会用尽一切办法混淆这一事实的。

我在选择证书厂商时会考虑下面四点。

- *客户支持*

  如果你的证书有问题，不管是浏览器支持（如果你的证书被客户的浏览器标记为不可信，你的客户会告诉你的）、安装问题，还是更新困扰，你都会感激良好的客户支持。这是你会买更贵的证书的原因之一。托管服务提供商一般会转售证书，并且根据我的经验，他们会提供更高级别的客户支持，因为他们还想留住托管客户。

- *避免链式根证书*

  证书链很普遍，也就是说你实际上要请求多个证书来建立安全连接。链式证书会导致额外的安装工作，因此我会多花点儿钱购买依赖于单个根证书的证书。一般很难（或不可能）确定你得到的是什么，这是寻找良好客户支持的另一个原因。如果你询问根证书是否是链式的，而他们不能或者不愿告诉你，你应该到别处看看。

- *单域证书、多子域证书、多域证书及通配证书*

  最便宜的证书一般是单域的。它们可能听上去不赖，但要记住，那意味着如果你给 meadowlarktravel.com 买了个证书，则它不能用于 www.meadowlarktravel.com，反之亦然。因此我一般尽量不用单域证书，尽管预算极其有限时它可能是个不错的选择（你总能设置好重定向，把请求导向恰当的域）。多子域证书的好处在于你可以购买一个证书来覆盖 meadowlarktravel.com、www.meadowlark.com、blog.meadowlarktravel.com、shop.meadowlarktravel.com 等多个域名。不足之处在于你必须预先知道你想用哪些子域名。如果你预见到自己将在一年内增加或使用不同的子域名（需要支持 HTTPS 的），使用通配证书可能会更好，它们一般会更贵。但它们能作用于任何子域名，并且你根本不需要指出子域名是什么。最后还有多域名证书，像通配证书一样，一般会更贵。这些证书支持整个多域名，比如说，你可以有 meadowlarktravel.com、meadowlarktravel.us、meadowlarktravel.com 和 www 的变体。

- *域证书、组织证书和扩展验证证书*

  有三种证书：域证书、组织证书和扩展验证证书。域证书，就像它的名字一样，只是证明你是在用你自己所认为的域名做业务。组织证书在某种程度上为你在打交道的真正组织提供保证。它们更难获得：通常会涉及书面工作，并且你必须提供省（州）和 / 或者

联邦商业名称记录、实际地址等信息。不同的证书厂商会要求不同的文件，所以一定要问问厂商要得到这些证书需要什么。最后是*扩展认证证书*，这是 SSL 证书中的劳斯莱斯。它们像组织证书一样能证实组织的存在，但它们要求更高标准的证据，甚至要求昂贵的审查来建立你的数据安全实践（尽管看起来这种要求越来越少了）。它们可能单个域名最少收 150 美元。我推荐你用不太昂贵的域名证书或者用扩展验证证书。组织证书尽管会证实组织的存在，但在浏览器中显示时没有任何差别，所以按我的经验，除非用户真去检查证书（非常罕见），否则它和域名证书没有明显的不同。而另一方面，扩展验证证书一般会向用户出示一些线索，表明他们所做的业务是合法的（比如 URL 栏是绿色的，并且组织名称显示在 SSL 图标旁边）。

如果你接触过 SSL 证书，可能会想我为什么没提到证书保险。我忽略了那个价格上的差异，因为它所担保的基本上是不可能发生的事。其核心思想是如果有人因为在你的网站上交易遭受了经济上的损失，并且他们能证明那些损失是因为加密不充分导致的，那么保险公司会出面承担你的损失。尽管确实有这种可能性，如果你的应用程序涉及财务交易，有人可能会因为经济损失对你采取法律行动，但因为加密不充分造成这种情况的可能性几乎为零。如果我因为链接到某个公司的在线服务遭受了经济损失，在我找这个公司想要挽回损失时，试图证明 SSL 加密被攻破绝对是我的最后一个选择。如果你要从两个只是价格和保险范围不同的证书中选一个，买便宜的那个。

买了证书后，你就可以进入一个安全区域下载你的私钥和证书（你可能要仔细检查下载链接本身是通过 HTTPS 协议的：通过一个未经加密的通道传输私钥是不明智的！）。别理那些想要通过邮件给你发私钥的证书厂商：邮件不是安全渠道。私钥的标准扩展名是 .pem，有时是 .key。证书的扩展名有 .crt、.cer 或 .der（证书的格式叫作“特异编码规则”，即 Distinguished Encoding Rules 或 DER，因此 .der 扩展名不太常见）。

18.1.4 对你的Express应用启用HTTPS

只要有了私钥和证书，在应用里使用它们很容易。让我们再回顾一下我们是如何创建服务器的：

```
http.createServer(app).listen(app.get('port'), function(){
        console.log('Express started in ' + app.get('env') +
                ' mode on port ' + app.get('port') + '.');
});
```

切换到 HTTPS 很简单。我建议你把私钥和 SSL 证书放在 ssl 子目录下（尽管放在项目根目录下的情况十分常见）。然后用 `https` 模块代替 `http` 模块，把 `options` 对象传给 `createServer` 方法就可以了：

```
var https = require('https'); // 一般在文件顶部

var options = {
```

```
        key: fs.readFileSync(__dirname + '/ssl/meadowlark.pem');
        cert: fs.readFileSync(__dirname + '/ssl/meadowlark.crt');
};

https.createServer(options, app).listen(app.get('port'), function(){
        console.log('Express started in ' + app.get('env') +
                ' mode on port ' + app.get('port') + '.');
});
```

就是这样。假如你用的还是 3000 端口，现在可以连接到 https://localhost:3000。如果你试着连接 http://localhost:3000，只会访问超时。

18.1.5　关于端口的说明

不管你知道与否，当你访问网站时，总是会连接到特定的端口上，即便在 URL 中没有指定也是这样的。如果没有指明端口，浏览器会假定 HTTP 用的是端口 80。实际上，如果你明确指定端口 80，大多数浏览器也不会显示这个端口号。比如说，访问 http:// www.apple.com:80，在页面加载时浏览器会把 :80 去掉。但它还是会连接端口 80，只是隐含的。

同样，HTTPS 的标准端口是 443。浏览器的处理也是一样的：如果你连接 https://www.google.com:443，大多数浏览器都不会显示 :443，但它们连的就是那个端口。

如果你的 HTTP 用的不是端口 80，HTTPS 不是 443，则必须明确指定端口和协议以保证正确连接。在同一个端口上连接 HTTP 和 HTTPS 是不可能的（从技术上来讲是有可能的，但没理由要这样做，而且实现起来非常复杂）。

如果你要在端口 80 上运行 HTTP，或者在 443 上运行 HTTPS，则不必明确指定端口，你只要考虑两点。第一个是很多系统已经有运行在端口 80 上的默认服务器了。比如说，如果你用的是 OS X，并且启用了 Web 共享，Apache 会运行在端口 80 上，你将无法在端口 80 上启动应用。

另一个要了解的是在大多数操作系统上，端口 1~1024 需要提升权限才能打开。比如在 Linux 和 OS X 机器上，如果你试图在端口 80 上启动应用，会因为 EACCES 错误而失败。要在端口 80 或 443（或者任何低于 1025 的端口）上运行，你需要用 `sudo` 命令提升权限。如果你没有管理员权限，就无法直接在端口 80 或 443 上启动服务器。

除非你管理的是自己的服务器，否则可能没有托管账户的 root 权限。那你想用 80 或 443 端口时怎么办呢？一般托管服务提供商会有以提升权限运行的某种代理服务，会将请求传给你那个运行在非特权端口上的应用。我们会就此话题在下一节展开更多讨论。

18.1.6　HTTPS和代理

正如我们所看到的那样，在 Express 中使用 HTTPS 非常容易，对于开发来说它工作得很

好。然而当你想要扩展网站来处理更多流量时，你会想要使用 Nginx 这样的代理服务器（见第 12 章）。如果你的网站运行在一个共享的托管环境下，几乎可以肯定的是，会有个代理服务器将请求路由给你的应用程序。

如果用了代理服务器，客户端（用户的浏览器）会跟代理服务器通信，不是直接连到你的服务器上。然后代理服务器很可能通过常规的 HTTP 跟你的应用通信（因为你的应用和代理服务器都运行在同一个可信网络中）。你经常会听到有人说 HTTPS 止于代理服务器。

在大多数情况下，只要你或你的托管服务提供商正确配置好代理服务器，让它处理 HTTPS 请求，你就不需要做任何额外的工作了。但如果你的应用程序需要同时处理安全和非安全请求则是个例外。

这个问题有三种解决办法。第一种是简单地将代理配置成所有 HTTP 请求都重定向到 HTTPS，本质上是强制所有跟你的应用程序的通信都通过 HTTPS。这种方式越来越常见了，并且肯定是一种简单的解决方案。

第二种方式在某种程度上是将客户端 - 代理所用的通信协议发给你的服务器。最常用的办法是通过 `X-Forwarded-Proto` 头。比如在 Nginx 中设定这个请求头：

```
proxy_set_header X-Forwarded-Proto $scheme;
```

然后在你的应用中检查用的是不是 HTTPS 协议：

```
app.get('/', function(req, res) {
        // 下面这行代码本质上等同于：if(req.secure)
        if(req.headers['x-forwarded-proto']==='https') {
                res.send('line is secure');
        } else{
                res.send('you are insecure!');
        }
});
```

> 在 Nginix 中有一个专门针对 HTTP 和 HTTPS 的服务器配置块。如果你在跟 HTTP 对应的配置块中设置 `X-Forwarded-Protocol` 时失败了，那客户端就可以伪造请求头欺骗你的应用程序，即便连接不是安全的，也能让你的应用程序认为是安全的。如果你用这种办法，一定要确保设置好 `X-Forwarded-Protocol` 请求头。

Express 提供了一些便利的属性，在你使用代理时改变行为（十分正确）。不要忘了用 `app.enable('trust proxy')` 告诉 Express 要相信代理。一旦你这样做了，`req.protocol`、`req.secure` 和 `req.ip` 将会指向客户端到代理的连接，不是到你的应用的。

18.2　跨站请求伪造

跨站请求伪造（CSRF）攻击利用了用户一般都会相信浏览器并且在同一个会话中访问多个网站这样的事实。在 CSRF 攻击中，恶意站点上的脚本会请求另外一个网站：如果你在另一个网站上登录过，恶意网站可以成功访问那个网站上的安全数据。

要防范 CSRF 攻击，你必须想办法确保请求合法地来自你的网站。我们的做法是给浏览器传一个唯一的令牌。当浏览器提交表单时，服务器会进行检查，以确保令牌是匹配的。`csurf` 中间件负责令牌的创建和验证；你只需要确保令牌包含在到服务器的请求中。安装 `csurf` 中间件（`npm install --save csurf`），然后引入它，添加一个令牌到 `res.locals` 中：

```
// 这个必须放在 cookie-parser 和 connect-session 的引入之后
app.use(require('csurf')());
app.use(function(req, res, next){
        res.locals._csrfToken = req.csrfToken();
        next();
});
```

`csurf` 中间件添加了 `csurfToken` 方法到请求对象上。我们不一定非要把它赋给 `res.locals`，可以将 `req.csurfToken()` 直接传给需要它的视图，但这个工作量一般更小。

现在你所有的表单（以及 AJAX 调用）都必须提供一个叫作 `_csrf` 的域，它必须跟生成的令牌相匹配。我们看一下怎么把它添加到表单中：

```
<form action="/newsletter" method="POST">
        <input type="hidden" name="_csrf" value="{{_csrfToken}}">
        名称：<input type="text" name="name"><br>
        邮箱：<input type="email" name="email"><br>
        <button type="submit"> 提交 </button>
</form>
```

中间件 `csurf` 会处理剩下的工作：如果 `body` 中的域没有有效的 `_csrf` 域，它会引发一个错误（确保你的中间件里有错误路由！）。你可以去掉隐藏域看看会发生什么。

> 如果你有一个 API，很可能不想让 `csurf` 中间件干扰它。如果你想限制其他网站访问这个 API，应该看看 `connect-rest` 的 API key 功能。要防止 `csurf` 干扰你的中间件，就在引入 `csurf` 之前引入它。

18.3　认证

认证是一个复杂的大课题。可惜大多数真正的 Web 应用程序都少不了认证部分。我能给你的最重要的经验是别试图自己做这个。如果你的名片上没有“安全专家”这样的头衔，可能还不清楚设计一个安全认证系统需要怎样复杂周密的思考。

注意，我不是说你不要试图认识自己应用程序中的安全系统，我只是建议你别试图自己构建它。你可以研究我推荐的开源认证技术的源码，那肯定能让你了解到为什么你不应该独自承接这样的任务。

18.3.1 认证与授权

尽管这两个词经常交叉使用，但其实它们之间有些细微的差别。*认证*是指验证用户的身份，即他们是自己所宣称的人。*授权*是指确定用户有哪些权力，可以访问、修改或查看什么。比如说，客户可能会授权允许访问他们的账号信息，草地鹨旅行社的员工得到授权可以访问其他人的账号信息或销售记录。

一般来说（但不总是这样），先认证，然后确定授权。授权可能非常简单（授权 / 没有授权）、宽泛（用户 / 管理员），或非常细化，指定不同账号类型的读、写、删除和更新权限。授权系统的复杂性取决于你所写的应用程序类型。

因为授权严重依赖于应用程序的细节，本书中我会使用非常广泛的验证方案（客户 / 员工），只提供一个粗略的轮廓。

我将经常使用缩写“auth”，但只在从上下文中可以明确知道它指的是“authentication”（认证）还是“authorization”（授权），或者说指的是谁根本无所谓时。

18.3.2 密码的问题

密码的问题在于每一个安全系统的强度取决于它最弱的环节。密码是要求用户提供的——这就是最弱的一环。人类是不善于想出安全密码的。在我写这本书的时候，2013 年的一次安全漏洞分析会上指出，最流行的密码是 12345，排在第二位的是 password（上一年它是第一位）。即便在 2013 年这样一个安全意识年，人们仍然选择了糟得可怜的密码。密码策略有要求，比如要有一个大写字母、一个数字和一个标点符号，结果人们用的密码是 Password1!。

甚至于分析常见密码列表对解决这个问题也帮助不大。人们开始把高质量的代码记在记事本里，放在电脑中未经加密的文件里，或者把它们通过邮件发给自己。

最终它一定会变成你的问题，应用设计者对此无能为力。然而，你可以做些事情来提升用户密码的安全性。一种做法是推掉责任，交给第三方做认证。另一种做法是把你的登录系统做得对密码管理服务更加友善，比如 LastPass、RoboForm 和 PasswordBox。

18.3.3 第三方认证

互联网上几乎每个人都至少有一个主流服务的账号，比如谷歌、Facebook、Twitter 或 LinkedIn，第三方认证正是借助了这一点。所有这些服务都提供了一种通过它们的服务认

证和识别用户的机制。

> 第三方认证经常被称为联合认证或代理认证。这些术语都可以换着用，尽管联合认证通常跟安全断言标记语言（SAML）和 OpenID 有关联，代理认证一般是跟 OAuth 关联的。

第三方认证有三个主要的优势。首先，你的认证负担降低了。你不用为认证单个用户操心，只要跟信任的第三方交互就行了。第二个优势是它能减轻“密码疲劳”，它是由太多账号引起的压力。我用 LastPass（http://lastpass.com），并且我刚查了我的密码保险箱，里面几乎有 400 个密码。作为一名技术专家，我的密码数量可能比互联网用户的平均数高，但一般大多数互联网用户也有几十甚至上百个账号。最后，第三方认证“没有摩擦”：用户可以用他们已有的账号更快地用上你的网站。如果用户发现他们还要再建一个用户名和密码，经常会选择离开。

如果你不用密码管理器，一般就是在大部分网站上使用相同的密码（大多数人都有个“安全的”密码用在银行之类的地方，还有一个“不安全的”密码用在其他所有地方）。这种方式有个问题，即你用的那些网站中只要有一个被攻破了，你的密码就暴露了，黑客会尝试用相同的密码访问其他服务。这就像把所有鸡蛋放在同一个篮子里。

第三方认证有它的缺点。尽管很难相信，但确实有人没有谷歌、Facebook、Twitter 或 LinkedIn 账号。然后在有这些账号的人中，怀疑（或想要保护隐私的想法）会让他们不愿意用那些账号登录你的网站。很多网站解决这个问题的方法是鼓励人们使用已有账号，但那些没有（或者不愿用）已有账号的人，可以为你的服务创建新的登录账号。

18.3.4　把用户存在数据库中

不管你是否依赖第三方认证用户，你都会想要在自己的数据库中保存一份用户记录。比如，你用 Facebook 做认证，那只是证实了用户的身份。如果你需要保存针对那个用户的配置信息，不可能也用 Facebook，你必须把跟那个用户相关的信息保存在你自己的数据库中。还有，你可能要让用户关联一个邮箱地址，并且他们可能不想用在 Facebook（或者你用的其他第三方服务）上用的那个。最后，在你的数据库中保存用户信息，你就可以自己做认证了，你应该想提供那一选择的。

那么我们给用户创建一个模型吧，`models/user.js`：

```
var mongoose = require('mongoose');

var userSchema = mongoose.Schema({
        authId: String,
        name: String,
```

```
        email: String,
        role: String,
        created: Date,
});

var User = mongoose.model('User', userSchema);
module.exports = User;
```

回想一下，MongoDB 数据库中的每一个对象都有自己唯一的 ID，存在它的 _id 属性中。然而那个 ID 是受 MongoDB 控制的。我们要想办法将用户记录映射到第三方 ID 上，所以我们有自己的 ID 属性，即 authId。因为我们用了多个认证策略，所以为了防止冲突，ID 是策略类型和第三方 ID 的组合。比如说，一个 Facebook 用户的 authId 是 facebook:525764102，而一个 Twitter 用户的 authId 是 twitter:376841763。

我们将在例子中使用两种角色："客户"和"员工"。

18.3.5　认证与注册和用户体验

认证是指通过可信的第三方或者你之前提供给用户的凭据（比如用户名和密码），验证用户的身份。注册是用户从你的网站上获取账号的过程（从我们的角度来看，注册是我们在数据库中给用户创建 User 记录的时刻）。

用户第一次加入你的网站时，应该让他们清楚自己正在注册。使用第三方认证系统时，我们可以在用户不知情的情况下完成注册，当然，前提条件是他们成功通过了第三方认证。这样做一般不太好，你应该让用户清楚他们在你的网站上注册了（不管他们是否通过第三方认证），并且提供一种清晰的机制让他们可以取消会员身份。

"第三方混乱"是要考虑的一个用户体验状况。如果用户在一月份用 Facebook 注册了你的服务，然后在七月份回来了。当面对一个有着 Facebook、Twitter、谷歌或 LinkedIn 登录选择的界面时，用户很可能已经忘了原来注册用的是哪个服务了。这是第三方认证的一个缺点，并且你对此几乎做不了什么。这是要求用户提供邮箱地址的另一个好理由：这样你就可以让用户通过邮箱找回账号，并且发送一封邮件给那个地址说明当初是用哪个服务认证的。

如果你觉得自己对用户所用的社交网络有充分的认识，可以提供一个"主认证服务"来缓解这个问题。比如，如果你十分肯定你的大部分用户都有 Facebook 账号，你可以在网站上放一个大按钮，写上"用 Facebook 登录"。然后用比较小的按钮，甚至只是用文本写上"或者用谷歌、Twitter 或 LinkedIn 登录"。这种方式可以减少第三方混乱情况出现的次数。

18.3.6　Passport

Passport 是为 Node/Express 做的认证模块，非常健壮，也非常流行。它没有绑死在任何认

证机制上，而是基于可插拔认证策略的思想（如果你不想用第三方认证，它也有本地策略）。理解认证信息流可能过于复杂了，所以我们先从一种认证机制入手，然后再添加更多的认证机制。

用第三方认证要明白的重要细节是你的应用绝对不会收到密码。完全是由第三方处理的。这是好事：第三方承担了安全处理和密码存储的重担。[1]

然后整个过程要靠重定向完成（如果你的应用程序接收不到用户的第三方密码，必须如此）。一开始你可能会觉得疑惑，你为什么能将本地服务器的 URL 传给第三方，而且还可以成功认证呢（毕竟第三方服务器在处理你的请求时并不知道你的本地服务器在哪里）？这之所以能实现，是因为第三方只是告知你的浏览器让它重定向，而你的浏览器处于你的网络中，因此可以重定向到本地地址。

基本流程如图 18-1 所示。这张图展示了功能上的重要流程，我们能很清楚地看到在第三方网站上实际发生的认证过程。好好享用这张图的简单明了吧，实际上事情比这复杂得多。

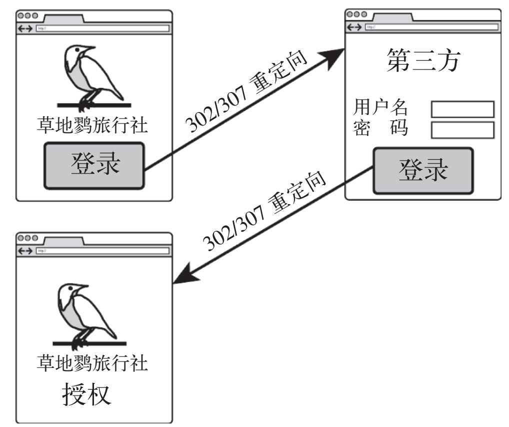


图 18-1　第三方认证流程

在使用 Passport 时，你的应用要负责四步。研究一下更详细的第三方认证流程，如图 18-2 所示。

注 1：第三方也不太可能存储密码。密码一般是通过存着的加盐哈希码（Salted Hash）进行验证的，这是对密码的单向转换。也就是说你只能从密码生成哈希码，但无法从哈希码中恢复出密码。对哈希码加盐可以对某些攻击进行额外的防护。

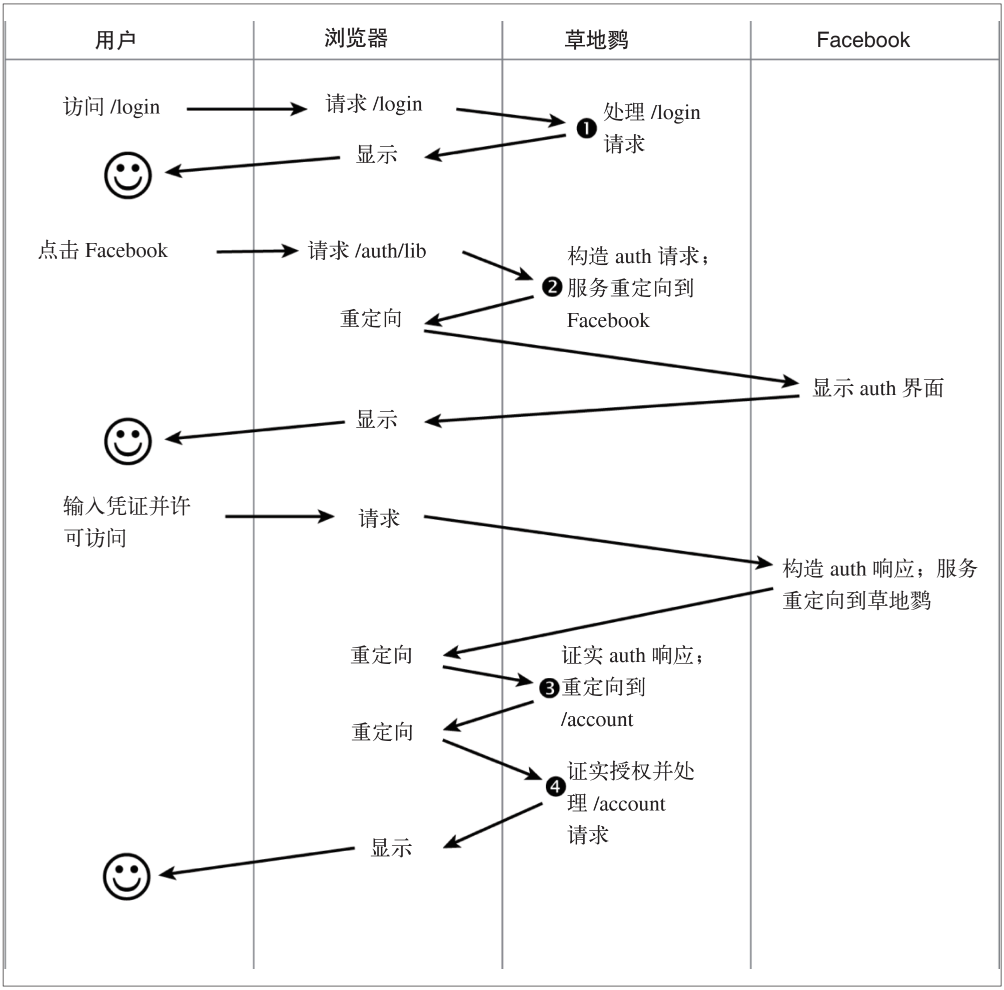


图 18-2　第三方认证流程细节图

简单起见，我们用草地鹨旅行社代表你的应用，Facebook 代表第三方认证机制。图 18-2 阐明了用户如何从登录页面到安全的“账号信息”页面（“账号信息”只是为了说明问题，它可以是你网站上任何需要认证的页面）。

这张图给出了你一般不会想到的细节，但重要的是要理解其所在的上下文。具体来说，当你访问一个 URL 时，并不是你在向服务器发起请求，实际上是浏览器发起的。也就是浏览器可以做三件事：发起 HTTP 请求、显示响应、执行重定向（这是发起另一个请求和显示另一个响应所必需的，然后可能是另一次重定向）。

在“草地鹨”那一栏，你可以看到你的应用程序真正要负责的那四步。好在我们是用 Passport（以及可插入的策略）来执行这些步骤的具体动作，否则这本书就要厚得多了。

在开始探讨实现细节之前，我们再稍微详细地考虑一下每一步。

- 登录页

  用户在登录页上可以选择登录的方法。如果你用的是第三方认证，通常那只是个按钮或链接。如果你用的是本地认证，它会有用户名和密码域。如果用户没有登录就试图访问一个需要认证的 URL（比如我们的 /account），这可能就是你要重定向的页面（或者你也可以重定向到一个有登录页面链接的“未经授权”页）。

- 构建认证请求

  在这一步，你要构造一个发送给第三方的请求（通过重定向）。这个请求的细节比较复杂，并且是专门针对这个认证策略的。Passport（和策略插件）会完成所有繁重的工作。这个认证请求会保护你免受“中间人”攻击，以及可能遭受的其他攻击。认证请求一般都很短命，所以你不能把它存下来指望以后再用：这也是一种保护措施，限制攻击者能够采取行动的时间窗口。你可以在这一步里向第三方授权机制请求更多信息。比如说，通常会请求用户名，可能还有邮箱地址。记住，你请求的用户信息越多，他们越不愿意给你的应用程序授权。

- 证实认证响应

  假定用户授权了你的应用程序，你就会从第三方得到一个有效的认证响应，即用户身份的证据。复杂的校验细节还是由 Passport（及策略插件）处理。如果认证响应表明用户没有授权（如果用户输入了无效的凭证，或者用户没有给你的应用程序授权），你会被重定向到一个合适的页面（或者回到登录页面，或者到“未经授权”页面，或者是“无法授权”页面）。在认证响应中会有用户在第三方的唯一 ID，以及你在第二步中请求的所有细节。要完成第四步，我们必须“记住”用户是授权过的。一般是设定一个包含用户 ID 的会话变量，表明这个会话已经经过授权了（也可以用 cookie，不过我建议用会话）。

- 证实授权

  在第三步中，我们在会话中保存了用户 ID。有了用户 ID，我们就可以从数据库中获取用户对象，得到其中包含的用户授权信息。按照这种方式，我们没必要每个请求都让第三方认证（那样会造成缓慢而痛苦的用户体验）。这个任务简单，我们也不需要 Passport 了：我们自己有包含认证规则的自有 `User` 对象（如果没有那个对象，表明请求没有授权，我们可以转到登录或“未经授权”页）。

> 用 Passport 实现认证需要做大量的工作，一会儿你就能看到了。然而认证是应用程序中的重要组成部分，我认为花些时间做好它是明智之举。还有些像 LockIt（http://www.mircozeiss.com/lockit-an-express-authentication-solution）这种试图提供更加“成品化”方案的项目。然而要充分发挥 LockIt（或类似方案）的作用，你必须明白认证和授权的细节，本章就是因此而生的。另外，如果你需要定制一个认证方案，Passport 是极佳的切入点。

搭建Passport

简单起见，我们还是从单个认证提供者开始。我们选择了 Facebook。在把 Passport 和 Facebook 策略搭建起来之前，我们还需要在 Facebook 里做点儿配置。要进行 Facebook 认证，你需要一个 Facebook 应用。如果你已经有合适的 Facebook 应用了，可以用那个，或者专门为认证新建一个。如果有可能，你应该用你们组织的官方 Facebook 账号创建这个应用程序。也就是说，如果你为草地鹨旅行社工作，那就用草地鹨旅行社的 Facebook 账号创建这个应用（为了方便管理，你随时可以将自己的个人 Facebook 账号添加为管理员）。如果出于测试目的，用你自己的的 Facebook 账号也没关系，但把个人账号用在生产环境中是不专业的表现，也会让用户产生怀疑。

Facebook 应用管理的细节似乎经常发生变化，所以我就不在这里展开讲了。如果你需要了解创建和管理应用的细节，请参考 Facebook 开发人员文档（https://developers.facebook.com/docs）。

为了进行开发和测试，你需要把开发 / 测试域名跟应用关联上。Facebook 允许你用本地服务器（和端口号），这对测试非常有利。此外，你也可以指定一个本地 IP 地址，如果你用虚拟服务器，或者你的网络上的另一台服务器测试会很有帮助。重要的是，你为了测试输入到浏览器中的 URL（比如 http://localhost:3000）是跟 Facebook 应用关联的。目前你只能给应用关联一个域名。如果你需要用多个域名，必须创建多个应用（比如你可以有 Meadowlark Dev、Meadowlark Test 和 Meadowlark Staging；生产应用可以简单地叫作 Meadowlark Travel）。

配置好应用后，你需要它的唯一 ID 和密钥，这两个都能在 Facebook 的应用管理页面上找到。

> 你将可能遭受的最大挫折之一是收到 Facebook 发来这样的消息："给定的 URL 不被应用程序配置允许。"这表明回调 URL 中的主机名和端口跟你在应用中的配置不一致。如果你看你浏览器中的 URL，你见到的将是编码后的 URL，你可以以此为线索。比如说，如果我用了 192.168.0.103:3443，并且收到了那条消息，我查看了 URL。如果在查询字符串中见到 `redirect_uri=https%3A%2F%2F192.68.0.103%3A3443%2Fauth%2Ffacebook%2Fcallback`，我很快就能发现错误：我的主机名中用的不是 168，而是 68。

现在我们装上 Passport 和 Facebook 认证策略：

```
npm install --save passport passport-facebook
```

在我们搞定之前还有很多认证代码（特别是要支持多种策略时），我们不想用这些代码弄乱 meadowlark.js。所以我们要创建一个 lib/auth.js 模块。这将会是一个大文件，我们

准备一块块儿地来。我们先从引入模块和 Passport 要求的两个方法开始，它们分别是 serializeUser 和 deserializeUser：

```
var User = require('../models/user.js'),
        passport = require('passport'),
        FacebookStrategy = require('passport-facebook').Strategy;

passport.serializeUser(function(user, done){
        done(null, user._id);
});

passport.deserializeUser(function(id, done){
        User.findById(id, function(err, user){
                if(err || !user) return done(err, null);
                done(null, user);
        });
});
```

Passport 用 serializeUser 和 deserializeUser 将请求映射到认证用户上，允许你使用任何存储方法。在这个例子中，我们只在会话中存放 MongoDB 赋予的 ID（User 模型实例的 _id 属性）。我们这样做让“序列化”（serialize）和“反序列化”（deserialize）有点儿名不副实了：实际上只在会话里存了一个用户 ID。然后当我们有需要时，会从数据库中查找那个 ID 得到 User 模型的实例。

实现了这两个方法后，只要有活跃的会话，并且用户成功通过认证，req.session.passport.user 就会对应上 User 模型的实例。

接下来我们要选择输出什么。为了启用 Passport 的功能，我们需要做两件事：初始化 Passport 并注册处理认证以及从第三方认证服务重定向的回调的路由。我们不想把它们两个合到一个函数中，因为在主程序文件中，我们可能想要选择把 Passport 连接到中间件链条中的时机（记住，在添加中间件时，顺序很重要）。所以我们不准备让模块输出函数做这些事情，而是要让它返回一个函数，这个函数返回的对象中有我们需要的方法。一开始为什么不只是返回一个对象呢？因为我们需要植入一下配置值。此外，因为我们需要将 Passport 中间件连入我们的应用程序，所以函数比较容易传入 Express 应用程序对象中：

```
module.exports = function(app, options){

        // 如果没有指定成功和失败的重定向地址
        // 设定一些合理的默认值
        if(!options.successRedirect)
                options.successRedirect = '/account';
        if(!options.failureRedirect)
                options.failureRedirect = '/login';

        return {

                init: function() { /* TODO */ },
```

```
                registerRoutes: function() { /* TODO */ },

        };
};
```

在探讨 `init` 和 `registerRoutes` 方法的细节之前，我们先看看将会如何使用这个模块（希望那样能让返回一个返回对象的函数这件事更清楚一点儿）：

```
var auth = require('./lib/auth.js')(app, {
        providers: credentials.authProviders,
        successRedirect: '/account',
        failureRedirect: '/unauthorized',
});
// auth.init() 链入了 Passport 中间件：
auth.init();

// 现在可以指定我们的 auth 路由了：
auth.registerRoutes();
```

注意，除了指定成功和失败时的重定向路径，我们还指定了一个 `providers` 属性，我们已经把它抽离到 credentials.js 文件中了（见第 13 章）。我们还需要把 `authProviders` 属性添加到 credentials.js 中：

```
module.exports = {
        mongo: {
                //...
        },
        authProviders: {
            facebook: {
                development: {
                    appId: 'your_app_id',
                    appSecret: 'your_app_secret',
                },
            },
        },
}
```

注意看，我们把应用的细节放到属性 `development` 中了。这样我们可以同时说明开发和生产应用（还记得 Facebook 不允许你将一个应用程序关联到多个 URL 上吧）。

> 像这样把认证代码绑在模块里还有一个原因，即我们可以在其他项目里重用它。实际上已经有一些认证包做了我们在这里做的基础工作。然而理解其中发生的细节很重要，所以即便你最终用别人写的模块，这也能帮你了解认证流程中发生的一切。

现在来处理 `init` 方法：

```
init: function() {
    var env = app.get('env');
```

```
        var config = options.providers;

        // 配置 Facebook 策略
        passport.use(new FacebookStrategy({
            clientID: config.facebook[env].appId,
            clientSecret: config.facebook[env].appSecret,
            callbackURL: '/auth/facebook/callback',
        }, function(accessToken, refreshToken, profile, done){
            var authId = 'facebook:' + profile.id;
            User.findOne({ authId: authId }, function(err, user){
                if(err) return done(err, null);
                if(user) return done(null, user);
                user = new User({
                    authId: authId,
                    name: profile.displayName,
                    created: Date.now(),
                    role: 'customer',
                });
                user.save(function(err){
                    if(err) return done(err, null);
                    done(null, user);
                });
            });
        }));

        app.use(passport.initialize());
        app.use(passport.session());
    },
```

这段代码密度有点儿大，但实际上大部分都是 Passport 的套路化代码。其中比较重要的是在函数中传入 FacebookStrategy 实例的部分。当调用这个函数时（用户成功通过认证后），参数 profile 中有 Facebook 用户的信息。最重要的是它包含 Facebook ID。我们要用这个把 Facebook 账号跟我们自己的 User 模型关联起来。注意，我们在属性 authId 前面加了前缀 facebook: 作它的命名空间。尽管可能性很小，但这可以防止 Facebook ID 跟 Twitter 或 Google ID 冲突（我们还可以借此检查用户模型，看用户用的是什么认证方法，这个很有用）。如果数据库已经有这个 ID（命名空间一致）了，直接返回它就可以了（这时会调用 serializeUser，由它把 MongoDB ID 放到会话中）。如果没有返回用户记录，我们会创建一个新的 User 模型并把它存到数据库中。

我们最后要做的是创建 registerRoutes 方法（别担心，这个很短的）：

```
    registerRoutes: function(){
        // 注册 Facebook 路由
        app.get('/auth/facebook', function(req, res, next){
            passport.authenticate('facebook', {
                    callbackURL: '/auth/facebook/callback?redirect=' +
                            encodeURIComponent(req.query.redirect),
            })(req, res, next);
        });
```

```
        app.get('/auth/facebook/callback',passport.authenticate('facebook',
            { failureRedirect: options.failureRedirect },
            function(req, res){
                    // 只有认证成功才能到这里
                    res.redirect(303, req.query.redirect || options.successRedirect);
                    }
            ));
    },
```

现在我们有路径 /auth/facebook；访问这个路径的用户会被自动重定向到 Facebook 的认证界面上（这是由 `passport.authenticate('facebook')` 完成的），如图 18-1 第二步所示。注意，我们在这里覆盖了默认的回调 URL，因为我们想包含我们来自哪里的信息。因为我们让浏览器重定向到了 Facebook 做认证，所以也要有办法能回来。一旦用户在 Facebook 上做了授权，浏览器就会转回到你的网站上。具体来说就是到 /auth/facebook/callback（带着可选的 `redirect` 查询字符串，表明用户最初是从哪里来的）。在查询字符串上还有将要由 Passport 证实的认证令牌。如果 Passport 未能证实，浏览器会被重定向到 `options.failureRedirect`。如果证实成功，Passport 会调用 `next()`，回到你的应用中。注意中间件是如何链到 /auth/facebook/callback 的处理器中的：先调用的 `passport.authenticate`。如果它调用 `next()`，控制器又交给了你的函数，如果没有指定 `redirect` 查询字符串参数，那就会重定向到原始地址，或者 `options.successRedirect`。

> 省略 `redirect` 查询字符串参数能简化认证路由。如果你只有一个 URL 需要认证，这可能比较有诱惑性。然而实现这个功能最终还是方便，并且能提供更好的用户体验。毫无疑问，你之前肯定有过这样的体验：你发现了自己想要的页面，然后发现需要登录。你登录了，然后被重定向到默认页，你只能自己再退回到原来那个页面上。这种用户体验无法令人满意。

这个过程 Passport 变的“魔术”是将用户（在这个例子中只是 MongoDB 数据库中的用户 ID）存到会话中。这是好事，因为浏览器被重定向了，这是不同的 HTTP 请求：没有会话中的信息，我们根本没办法知道用户已经通过认证了！用户认证成功后，`req.session.passport.user` 就设定好了，后续的请求也就知道这个用户已经通过认证了。

我们看一下 `/account` 处理器，看它如何检查以确保用户是通过认证的（这个路由处理器将会出现在我们的主应用程序文件中，或者在单独的路由模块中，不会在 /lib/auth.js 中）：

```
    app.get('/account', function(req, res){
            if(!req.session.passport.user)
                    return res.redirect(303, '/unauthorized');
            res.render('account');
    });
```

现在只有已认证用户才能见到账号页面了，其他所有人都会被转到“未经授权”页面。

18.3.7 基于角色的授权

到目前为止，从技术上来讲我们还没做过任何授权（我们只是把已授权和未授权用户分开了）。然而，假设我们只想让客户见到他们自己的账号视图（员工能看到用户的账号信息，所以可能有不同视图的访问权限）。

记住，在单个路由中，你可以有多个函数，它们是按顺序调用的。我们创建一个 customerOnly 函数，只允许客户访问：

```
function customerOnly(req, res){
        var user = req.session.passport.user;
        if(user && req.role==='customer') return next();
        res.redirect(303, '/unauthorized');
}
```

再创建一个 employeeOnly 函数，和上一个函数稍有不同。比如说有个 /sales 路径，我们只想让员工访问。更进一步讲，我们甚至不想让外人知道有这样一个地址，即便他们不小心访问了这个地址。如果潜在的攻击者访问了 /sales，然后看到“未经授权”页面，这也是让攻击变得更容易的信息（只是知道这个页面在那儿）。所以为了增加一点儿安全性，当非员工访问 /sales 页面时，我们想让他们看到常规的 404 页面，让潜在的攻击者无从下手：

```
function employeeOnly(req, res, next){
        var user = req.session.passport.user;
        if(user && req.role==='employee') return next();
        next('route');
}
```

调用 next('route') 不是执行路由中的 next 处理器，它会跳过这个路由。如果接下来没有处理 /account 的路由，那最终就是到 404 处理器，和我们期望的结果一致。

看把这些函数用起来多简单：

```
// 客户路由

app.get('/account', customerOnly, function(req, res){
        res.render('account');
});
app.get('/account/order-history',customerOnly,function(req, res){
        res.render('account/order-history');
});
app.get('/account/email-prefs', customerOnly, function(req, res){
        res.render('account/email-prefs');
});

// 员工路由
app.get('/sales', employeeOnly, function(req, res){
        res.render('sales');
});
```

你应该清楚，基于角色的授权是简单还是复杂取决于你。比如说，如果你想允许多个角色访问会怎么样？你可以用下面的函数和路由：

```
function allow(roles) {
        var user = req.session.passport.user;
        if(user && roles.split(',').indexOf(user.role)!==-1) return next();
        res.redirect(303, '/unauthorized');
}

app.get('/account', allow('customer,employee'), function(req, res){
         res.render('account');
});
```

希望这个例子能对你有所启发，让你了解在基于角色的授权上你能发挥出什么样的创造性。你甚至可以基于其他属性授权，比如用户成为会员的时长，或者用户跟你预定了多少次旅行。

18.3.8 添加更多认证提供者

现在我们的框架已经搭好了，添加更多的认证服务提供者很容易。比如我们想用谷歌认证。对于谷歌，我们甚至不需要获取应用密钥或修改我们的 authProviders.js 文件。只要把下面的代码添加到 lib/auth.js 文件的 `init` 方法中：

```
passport.use(new GoogleStrategy({
    returnURL: 'https://' + host + '/auth/google/return',
    realm: 'https://' + host +'/',
}, function(identifier, profile, done){
    var authId = 'google:' + identifier;
    User.findOne({ authId: authId }, function(err, user){
        if(err) return done(err, null);
        if(user) return done(null, user);
        user = new User({
            authId: authId,
            name: profile.displayName,
            created: Date.now(),
            role: 'customer',
        });
        user.save(function(err){
            if(err) return done(err, null);
            done(null, user);
        });
    });
}));
```

并把下面的代码添加到 `registerRoutes` 方法中：

```
// 注册谷歌路由
app.get('/auth/google', function(req, res, next){
        passport.authenticate('google', {
                callbackURL: '/auth/google/callback?redirect=' +
```

```
                encodeURIComponent(req.query.redirect),
        })(req, res, next);
});
app.get('/auth/google/callback', passport.authenticate('google',
        { failureRedirect: options.failureRedirect },
        function(req, res){
                // 只有认证成功才能到这里
                res.redirect(303, req.query.redirect || options.successRedirect);
        }
));
```

18.4 小结

恭喜你完成了最复杂的一章！这样重要的一个功能特性（认证与授权）这么复杂实在是不幸，但在这样一个到处都充斥着安全威胁的世界，这种复杂性是不可避免的。好在有Passport 这样的项目（以及基于其上构建的优秀的认证方案）减轻了我们的负担。我还是要奉劝你不要对应用程序的这一领域掉以轻心，尽职尽责地磨练你在安全领域的技能会让你变成优秀的互联网公民。你的用户可能永远不会因此对你表示感谢，但因为糟糕的安全让用户的数据受到损害是应用程序所有者的耻辱。

第19章

集成第三方API

渐渐地，成功的网站不再是完全独立的了。为了留住已有用户，找到新用户，跟社交媒体集成是网站必须要做的工作。为了提供店铺定位或其他基于位置的服务，必须使用地理定位和地图服务。而且还不止于此：越来越多的组织意识到提供 API 有助于扩展他们的服务，并会让服务更实用。

本章会讨论两种最常见的集成需求：社交媒体和地理定位。

19.1　社交媒体

社交媒体对产品和服务的推广非常有帮助。如果这是你的目标，让你的用户能轻松地在社交媒体上分享你的内容是必需的功能。在我编写本书时，Facebook 和 Twitter 在社交网络服务中占据着统治性地位。Google+ 可能可以勉强分一杯羹，但完全无法跟他们抗衡：毕竟他们后面是世界上规模最大、最精明的互联网公司。像 Pinterest、Instagram 和 Flickr 这样的网站也有自己的一席之地，但通常他们都有特定受众（比如说，如果你的网站是关于 DIY 手工艺的，你应该一定想要支持 Pinterest）。如果你想笑就笑吧，但我预计 MySpace 还会回来的。他们重新设计的网站很赞，并且最重要的是用 Node 做的。

19.1.1　社交媒体插件和站点性能

大多数社交媒体集成都是前端事务。你在你的页面中引用恰当的 JavaScript 文件，它就能实现内容的流入（比如从你的 Facebook 页面上抓来三个头条）和内容的流出（比如就你所在的页面发送推文）。尽管这一般代表着社交媒体集成最容易的路径，但它也是有代价的：我

曾经见过因为额外的 HTTP 请求占用了两倍甚至三倍加载时间的页面。如果你很看重页面的性能（应该是这样，特别对于移动端用户来说），应该认真考虑一下如何集成社交媒体。

那就是说，实现 Facebook 的点“赞”按钮或“推文”按钮的代码利用了浏览器中的 cookie，它们以用户的名义提交内容。将这个功能挪到后端比较困难（并且在某些情况下是不可能的）。所以如果你需要这个功能，引入恰当的第三方库是你的最佳选择，即便它可能影响你的页面性能。Facebook 和 Twitter API 的普及程度非常高，你的浏览器很可能已经缓存了这些第三方库，这对性能也会有点儿帮助。

19.1.2　搜索推文

比如说我们想提到最近 10 条有标签 #meadowlarktravel 的推文。我们可以用一个前端组件实现这个功能，但那需要额外的 HTTP 请求。另外，如果我们在后端做，就可以缓存它以便提升性能。还有，如果在后端搜索，我们可以把严厉的推文“拉黑”，而这在前端很难实现。

Twitter 和 Facebook 一样，允许你创建应用。但它所谓的应用有点儿名不符实：Twitter 应用什么也不做（从传统意义来看）。它更像是一组凭据，你可以用它在你的网站上创建真正的应用。要访问 Twitter API，最容易的，并且也是可移植性最强的办法是创建一个应用，并用它获取访问令牌。

到 http://dev.twitter.com 创建 Twitter 应用。点击左上角的用户图标，选择“我的应用程序”。点击“创建一个新应用程序”，然后按照指示操作。应用程序创建好之后，你会看到一个消费者键和一个消费者密码。消费者密码如其名所示，应该保密：不要把它放到响应中发送给客户端。如果有第三方得到这个密码，他们就可以代表你的应用程序发起请求，如果他们恶意使用，会对你产生不良影响。

有了消费者键和消费者密码，我们就可以跟 Twitter REST API 通信了。

为了保持代码的整洁性，我们要把 Twitter 的代码放在模块 lib/twitter.js 中：

```
var https = require('https');

module.exports = function(twitterOptions){

        return {
                search: function(query, count, cb){
                        // TODO
                }
        };
};
```

你应该开始熟悉这种模式了。我们的模块输出了一个函数到传给配置对象的调用者中。这

个函数返回的是一个包含方法的对象。这样我们可以向模块中添加功能。目前我们只提供了一个 search 方法。下面是我们如何使用这个库的代码：

```
var twitter = require('./lib/twitter')({
        consumerKey: credentials.twitter.consumerKey,
        consumerSecret: credentials.twitter.consumerSecret,
});

twitter.search('#meadowlarktravel', 10, function(result){
        // 推文会在 result.statuses 中
});
```

（别忘了在 credentials.js 文件中放一个带 consumerKey 和 consumerSecret 的 twitter 的属性。）

在实现 search 方法之前，我们必须提供到 Twitter 的认证方法。过程很简单：基于消费者键和消费者密码用 HTTPS 请求一个访问令牌。这个只需要做一次：目前 Twitter 不会让访问令牌过期（尽管你可以手工让它们失效）。因为我们不想每次都请求访问令牌，所以要把它缓存起来以备后用。

构造模块的方式可以创建私有功能，让调用者无法访问。具体来说，调用者只能访问 module.exports。因为我们返回了一个函数，所以调用者只能访问那个函数。调用那个函数会得到一个对象，并且调用者只能访问那个对象的属性。所以我们准备创建一个变量 accessToken，用来缓存我们的访问令牌，还有一个函数 getAccessToken 用来获取访问令牌。第一次调用这个函数时，它会向 Twitter API 发起一个请求获取访问令牌。后续调用直接返回 accessToken 的值：

```
var https = require('https');

module.exports = function(twitterOptions){

        // 这个变量在模块外是不可见的
        var accessToken;

        // 这个函数在模块外是不可见的
        function getAccessToken(cb){
                if(accessToken) return cb(accessToken);
                // TODO: 获取访问令牌
        }

        return {
                search: function(query, count, cb){
                        // TODO
                },
        };
 };
```

因为 getAccessToken 可能需要异步调用 Twitter API，所以我们必须提供一个回调函数，等 accessToken 的值有效时调用。现在基本结构已经做好了，接下来实现 getAccessToken：

```
function getAccessToken(cb){
    if(accessToken) return cb(accessToken);

    var bearerToken = Buffer(
        encodeURIComponent(twitterOptions.consumerKey) + ':' +
        encodeURIComponent(twitterOptions.consumerSecret)
    ).toString('base64');

    var options = {
        hostname: 'api.twitter.com',
        port: 443,
        method: 'POST',
        path: '/oauth2/token?grant_type=client_credentials',
        headers: {
            'Authorization': 'Basic ' + bearerToken,
        },
    };

    https.request(options, function(res){
        var data = '';
        res.on('data', function(chunk){
            data += chunk;
        });
        res.on('end', function(){
            var auth = JSON.parse(data);
            if(auth.token_type!=='bearer') {
                console.log('Twitter auth failed.');
                return;
            }
            accessToken = auth.access_token;
            cb(accessToken);
        });
    }).end();
}
```

构造这个调用的详细信息请参考 Twitter 的开发者文档中的应用程序认证页面（https://dev.twitter.com/docs/auth/application-only-auth）。整个过程基本上是先基于消费者键和消费者密码组合构造一个 base64 编码的不记名令牌（bearer token）。这个令牌构造好之后，调用 /oauth2/tokenAPI，在 Authorization 请求头中包含不记名令牌，请求获取访问令牌。注意，这里必须用 HTTPS：如果你试图通过 HTTP 发起这个请求，那你的密钥就是未经加密传输的，API 会搁置你的请求。

得到 API 的完整响应后（监听响应流的 end 事件），解析 JSON，确保令牌类型是不记名，并且进展顺利。我们缓存访问令牌，然后调用回调函数。

现在有了获取访问令牌的机制，可以实现 API 调用了。所以接下来我们实现 search 方法：

```
search: function(query, count, cb){
        getAccessToken(function(accessToken){
                var options = {
                        hostname: 'api.twitter.com',
                        port: 443,
                        method: 'GET',
                        path: '/1.1/search/tweets.json?q=' +
                                encodeURIComponent(query) +
                                '&count=' + (count || 10),
                        headers: {
                                'Authorization': 'Bearer ' + accessToken,
                        },
                };
                https.request(options, function(res){
                        var data = '';
                        res.on('data', function(chunk){
                                 data += chunk;
                        });
                        res.on('end', function(){
                                 cb(JSON.parse(data));
                        });
                }).end();
        });
},
```

19.1.3 渲染推文

现在我们能搜索推文了。那如何在网站上显示它们呢？这主要取决于你，但还有些事情要考虑一下。Twitter 会确保对它的数据的使用跟其品牌保持一致。因此它确实有显示上的要求（https://dev.twitter.com/terms/display-requirements），你必须引入其功能元素显示推文。

这个要求有一定的灵活空间（比如，如果你要在一台不支持图片的设备上显示，就没必要包含头像），但总体来说，你最终得到的是跟嵌入式推文很像的东西。要做很多工作，也有办法绕过它，但它涉及连接到 Twitter 的小工具库，而这正是我们要极力避开的 HTTP 请求。

如果要显示推文，最好使用 Twitter 的小工具库，尽管它要发起额外的 HTTP 请求（但因为 Twitter 的普及程度，这个资源很可能已经被浏览器缓存下来了，所以对性能的影响应该可以忽略）。对于更加复杂的 API 应用，你还需要从后台访问 REST API，所以最终你很可能还要结合前端脚本使用 REST API。

继续之前的例子：我们想要显示标签为 #meadowlarktravel 的前十条推文。我们会用 REST API 搜索推文，用 Twitter 小工具库显示它们。因为我们不想碰到使用限制（或拖慢我们的服务器），所以会把推文和显示它们的 HTML 缓存 15 分钟。

我们一开始会修改 Twitter 库，引入 `embed` 方法，它会得到显示推文的 HTML（确保在文件顶部放了 `var querystring = require('query string');`）：

```
embed: function(statusId, options, cb){
    if(typeof options==='function') {
        cb = options;
        options = {};
    }
    options.id = statusId;
    getAccessToken(function(accessToken){
        var requestOptions = {
            hostname: 'api.twitter.com',
            port: 443,
            method: 'GET',
            path: '/1.1/statuses/oembed.json?' +
                querystring.stringify(options);
            headers: {
                'Authorization': 'Bearer ' + accessToken,
            },
        };
        https.request(requestOptions, function(res){
            var data = '';
            res.on('data', function(chunk){
                data += chunk;
            });
            res.on('end', function(){
                cb(JSON.parse(data));
            });
        }).end();
    });
},
```

现在推文的搜索和缓存都准备好了。我们在主应用程序文件中创建一个对象存储缓存：

```
var topTweets = {
        count: 10,
        lastRefreshed: 0,
        refreshInterval: 15 * 60 * 1000,
        tweets: [],
}
```

接下来我们会创建一个函数获取排在前面的推文。如果已经缓存了，并且缓存还没有过期，可以直接返回 `topTweets.tweets`。否则进行搜索，然后重复调用 `embed` 得到嵌入的 HTML。因为这是最后一块，所以我们准备介绍一个新概念：promise。promise 是一种管理异步功能的技术。异步函数不会立即返回，但我们可以创建一个 promise，异步部分一完成就马上 resolve。我们将会用到 Q promises 库（https://npmjs.org/package/q），所以你一定要运行 `npm install --save q`，并且把 `var Q = require(q);` 放在主应用程序文件顶部。下面是这个函数：

```
function getTopTweets(cb){
        if(Date.now() < topTweets.lastRefreshed + topTweets.refreshInterval)
                return cb(topTweets.tweets);
```

```
            twitter.search('#meadowlarktravel', topTweets.count, function(result){
                    var formattedTweets = [];
                    var promises = [];
                    var embedOpts = { omit_script: 1 };
                    result.statuses.forEach(function(status){
                            var deferred = Q.defer();
                            twitter.embed(status.id_str, embedOpts, function(embed){
                                    formattedTweets.push(embed.html);
                                    deferred.resolve();
                            });
                            promises.push(deferred.promise);
                    });
                    Q.all(promises).then(function(){
                            topTweets.lastRefreshed = Date.now();
                            cb(topTweets.tweets = formattedTweets);
                    });
            });
    }
```

如果你刚接触异步编程，这看起来可能会比较奇怪，所以我们花点时间分析一下这段代码。来看一个经过简化的例子，异步地对集合中的每个元素做些事情。

在图 19-1 中，有几个执行步数是我任意给定的。它们的任意在于第一个异步块可能是第 23 步或第 50 步，也可能是第 500 步，这取决于应用程序中有多少其他事情。同样，第二个异步块也可能在任何时刻执行（但感谢 promise，我们知道它一定是在第一个代码块之后）。



图 19-1　Promise

我们在第 1 步创建了个数组存放 promise，第 2 步开始循环遍历集合中的东西。注意，即便把函数放在 forEach 中，它也不是异步的：对集合中的每个元素同步调用函数，所以我们知道第 3 步在函数内部。第 4 步调用 api.async，它表示一个异步工作的方法。当它完成时，会调用你传入的回调函数。注意，console.log(num) 不会是第 4 步，因为异步函数还没机会完成并调用回调。相反，是第 5 行先执行（只是将我们刚刚创建的 promise 添加到数组中），然后再次开始（第 6 步和第 3 步是同一行）。迭代完成后（尽管 things 中还有很多东西），forEach 循环就结束了，然后第 6 步执行。第 6 步很特殊，它说："所有 promise 都 resolve 后执行这个函数。"本质上这是另一个异步函数，但它要等到我们对 api.async 的三次调用都完成后才会执行。第 7 步执行，向控制台输出些东西。所以即便代码中的 console.log(num) 在 console.log('other stuff...') 前面，也是 "other stuff " 先输出。第 13 步之后，"other stuff" 出现了。在某一点上，没什么事情要做了，JavaScript 引擎将会找些别的事情做。所以它去执行第一个异步函数：做完后调用回调函数，我们就到了第 15 步和第 16 步。那两步还会再重复，直到 things 中没有需要处理的元素。等所有 promise 都 resolve 后，那时候（并且只有等到那时候）就可以到 23 步了。

异步编程（和 promise）可能会需要你花点时间去消化，但它值得你认真研究：你会发现自己能用全新的、生产率更高的方式思考了。

19.2 地理编码

地理编码是指将街道地址或地名（布莱切利公园，西华道，布莱切利，米尔顿凯恩斯 MK3 6EB，英国）转换为地理坐标（纬度 51.9976597，经度 − 0.7406863）的过程。如果你的应用程序准备做地理位置计算（距离或方向），或者要显示地图，那你就需要地理坐标。

> 你可能习惯于用度、分、秒（DMS）表示地理坐标，但地理编码 API 和地图服务用单浮点数表示经纬度。如果你需要显示 DMS 坐标，请查阅 http://en.wikipedia.org/wiki/geographic_coordinate_conversion。

19.2.1 用谷歌的地理编码

谷歌和必应都有优秀的 REST 地理编码服务。我们以谷歌为例，但必应的服务跟它非常像。我们先创建模块 lib/geocode.js：

```
var http = require('http');

module.exports = function(query, cb){

        var options = {
                hostname: 'maps.googleapis.com',
                path: '/maps/api/geocode/json?address=' +
```

```
                encodeURIComponent(query) + '&sensor=false',
        };

        http.request(options, function(res){
                var data = '';
                res.on('data', function(chunk){
                        data += chunk;
                });
                res.on('end', function(){
                        data = JSON.parse(data);
                        if(data.results.length){
                                cb(null, data.results[0].geometry.location);
                        } else{
                                cb("No results found.", null);
                        }
                });
        }).end();
}
```

现在我们有了一个连接谷歌 API 对地址做地理编码的函数。如果它找不到地址（或因为其他原因失败了），会返回一个错误。谷歌 API 可以返回多个地址。比如说，如果你搜索“10 主大道”，但没指明城市、省或邮编，它会返回很多结果。我们的实现会直接取第一个。谷歌 API 返回很多信息，但我们目前只对坐标感兴趣。你可以根据自己的需要修改这个接口，让它返回更多信息，很容易的。查阅谷歌地理编码 API 文档（https://developers.google.com/maps/documentation/geocoding）了解 API 返回数据的更多信息。注意，在 API 请求中有一个 `&sensor=false`，这是个必填域，有位置传感器的设备应该设为 `true`，比如手机。你的服务器应该无法定位位置，所以它应该是 `false`。

使用限制

谷歌和必应对地理编码 API 的使用都有限制，以防止出现滥用的情况，但限制非常高。在编写本书时，谷歌的限制是每 24 小时不超过 2500 次请求。谷歌的 API 还要求你在网站上使用谷歌地图。也就是说，如果你用谷歌的服务对数据做地理编码，就不能转而将那些信息显示在必应的地图上，否则是违反服务协议的。一般来说，这并不算是一个严苛的限制，因为如果你不准备在地图上显示位置，也不会做地理编码。然而如果你更喜欢必应的地图，或者更喜欢谷歌的地图，就应该尊重它们的服务协议，选择恰当的 API。

19.2.2　对你的数据做地理编码

比如草地鹨旅行社正在通过代理商销售俄勒冈主题的产品（T 恤、马克杯之类），并且我们想在网站上加一个“寻找代理商”的功能，但我们没有代理商的坐标信息，只有街道地址。这时我们就要用地理编码 API。

在开始之前要考虑两件事情。初始化时数据库中可能已经有些代理商了。我们要批量处理这些代理商的地理编码。但将来新增代理商时，或者代理商地址发生变化时，怎么办呢？

这两种情况可以用相同的代码处理，但有些地方比较复杂。首先是使用限制。如果代理商数量超过 2500 个，我们就必须把初始化的地理编码工作分散到几天里完成，以避免触及谷歌的 API 限制。还有，初始化批量处理可能需要很长时间，我们不想让用户等一个小时甚至更长的时间才能看到代理商地图！然而完成初始化的批量地理编码后，新增加的代理商和修改地址的代理商可以零敲碎打地处理。我们先从代理商模型开始，在 models/dealer.js 中：

```
var mongoose = require('mongoose');

var dealerSchema = mongoose.Schema({
        name: String,
        address1: String,
        address2: String,
        city: String,
        state: String,
        zip: String,
        country: String,
        phone: String,
        website: String,
        active: Boolean,
        geocodedAddress: String,
        lat: Number,
        lng: Number,
});

dealerSchema.methods.getAddress = function(lineDelim){
        if(!lineDelim) lineDelim = '<br>';
        var addr = this.address1;
        if(this.address2 && this.address2.match(/\S/))
                addr += lineDelim + this.address2;
        addr += lineDelim + this.city + ', ' +
                this.state + this.zip;
        addr += lineDelim + (this.country || 'US');
        return addr;
};

var Dealer = mongoose.model("Dealer", dealerSchema);
module.exports = Dealer;
```

填充数据库（转换电子表格或手工录入）时可以先忽略 `geocodedAddress`、`lat` 和 `lng` 域。填充好数据库，可以着手处理地理编码的工作了。

我们准备采取跟 Twitter 缓存类似的办法。因为只缓存 10 条推文，所以我们把缓存放在了内存里。代理商信息可能大得多，并且为了提高速度仍然要缓存它，但不能放在内存里。然而我们想用一种在客户端超级快的办法，所以要用这些数据创建一个 JSON 文件。

接下来创建缓存：

```
var dealerCache = {
        lastRefreshed: 0,
        refreshInterval: 60 * 60 * 1000,
```

```
        jsonUrl: '/dealers.json',
        geocodeLimit: 2000,
        geocodeCount: 0,
        geocodeBegin: 0,
}
dealerCache.jsonFile = __dirname +
        '/public' + dealerCache.jsonUrl;
```

首先要创建一个辅助函数，对给定 Dealer 模型做地理编码，并将结果保存到数据库中。注意，如果当前代理商的地址跟最近一个地理编码匹配，则什么也不干直接返回。这样如果代理商的坐标是最新的，这个方法就非常快：

```
function geocodeDealer(dealer){
        var addr = dealer.getAddress(' ');
        if(addr===dealer.geocodedAddress) return; // 已经处理过了

        if(dealerCache.geocodeCount >= dealerCache.geocodeLimit){
                // 自上次做完地理编码已经过去 24 小时了吗？
                if(Date.now() > dealerCache.geocodeCount + 24 * 60 * 60 * 1000){
                        dealerCache.geocodeBegin = Date.now();
                        dealerCache.geocodeCount = 0;
                } else {
                        // 现在还不能做地理编码处理
                        // 我们已经达到使用限制了
                        return;
                }
        }

        geocode(addr, function(err, coords){
                if(err) return console.log('Geocoding failure for ' + addr);
                dealer.lat = coords.lat;
                dealer.lng = coords.lng;
                dealer.save();
        });
}
```

我们可以把 geocodeDealer 作为 Dealer 模型中的方法。但因为这个方法依赖于地理编码库，所以最好还是把它作为一个单独的函数。

现在可以创建一个函数刷新代理商缓存。这个操作可能需要些时间（特别是第一次执行时），但很快就能处理好：

```
dealerCache.refresh = function(cb){

        if(Date.now() > dealerCache.lastRefreshed + dealerCache.refreshInterval){
                // 我们要刷新缓存
                Dealer.find({ active: true }, function(err, dealers){
                        if(err) return console.log('Error fetching dealers: '+
                                err);
```

```
                        // 如果坐标是最新的，geocodeDealer 什么也不做
                        dealers.forEach(geocodeDealer);

                        // 现在将所有代理商写到缓存的 JSON 文件中
                        fs.writeFileSync(dealerCache.jsonFile, JSON.stringify(dealers));

                        // 搞定——调用回调
                        cb();
                });
        }
}
```

最后我们需要确立一个办法来及时更新缓存的数据。可以用 setInterval，但如果很多代理商发生了变化，有可能（如果可能性不太大）要花一个多小时刷新缓存。所以在刷新完成后用 setTimeout 等一个小时再刷新缓存：

```
function refreshDealerCacheForever(){
        dealerCache.refresh(function(){
                // 刷新间隔结束后调用自己
                setTimeout(refreshDealerCacheForever,
                        dealerCache.refreshInterval);
        });
}
```

> 我们没把 refreshDealerCacheForever 做成 dealerCache 的方法，因为 JavaScript 在处理 this 对象时很怪异。特别是当你调用一个函数（不是方法）时，this 不会绑定到调用对象的上下文上去。

现在终于可以启动我们的计划了。在第一次启动应用时，缓存还不存在，所以先创建一个空的，然后启动 dealerCache.refreshForever：

```
// 如果缓存还不存在，则创建它，以防出现 404 错误
if(!fs.existsSync(dealerCache.jsonFile))fs.writeFileSync(JSON.stringify([]));
// 开始刷新缓存
refreshDealerCacheForever();
```

注意，只有所有代理商数据都从数据库返回时才会更新缓存，任何需要地理编码的代理商都是如此。所以最坏的情况下，如果添加或更新了代理商，被更新的信息出现在网站上所需的时间是刷新间隔加上地理编码所需的时间。

19.2.3 显示地图

尽管显示代理商地图确实属于“前端”的工作，但做了这么多工作，如果看不到劳动果实也是挺扫兴的。所以我们准备稍微偏离一下本书的主题，做一些前端的工作，看看如何在

地图上显示我们刚做过地理编码的代理商。

跟地理编码 REST API 不同，在你的 Web 页面上用交互式的谷歌地图需要 API 密钥，也就是说你得有谷歌账号。谷歌 API 密钥文档（https://developers.google.com/maps/documentation/javascript/tutorial#api_key）里有如何获取 API 密钥的说明。

首先，我们加些 CSS 样式：

```
.dealers #map {
        width: 100%;
        height: 400px;
}
```

这样会创建一个适于移动设备的地图，宽度可以伸展到其所在容器的宽度，但高度是固定的。基本的样式有了，我们可以创建一个视图（views/dealers.handlebars），在地图上显示代理商以及代理商列表：

```
<script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY↲
&sensor=false"></script>
<script src="http://cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.3.0/↲
handlebars.min.js"></script>

<script id="dealerTemplate" type="text/x-handlebars-template">
    \{{#each dealers}}
        <div class="dealer">
            <h3>\{{name}}</h3>
            \{{address1}}<br>
            \{{#if address2}}\{{address2}}<br>\{{/if}}
            \{{city}}, \{{state}} \{{zip}}<br>
            \{{#if country}}\{{country}}<br>\{{/if}}
            \{{#if phone}}\{{phone}}<br>\{{/if}}
            \{{#if website}}<a href="{{website}}">\{{website}}</a><br>\{{/if}}
        </div>
    \{{/each}}
</script>

<script>
    var map;
    var dealerTemplate = Handlebars.compile($('#dealerTemplate').html());
    $(document).ready(function(){

        // 将地图的中心位置放在 US，设置缩放级别显示全国
        var mapOptions = {
            center: new google.maps.LatLng(38.2562, -96.0650),
            zoom: 4,
        };

        // 地图初始化
        map = new google.maps.Map(
            document.getElementById('map'),
            mapOptions);
```

```
            // 获取 JSON
            $.getJSON('/dealers.json', function(dealers){

                // 在地图上为每个代理商添加标记
                dealers.forEach(function(dealer){
                    // 跳过没有地理编码的代理商
                    if(!dealer.lat || !dealer.lng) return;
                    var pos = new google.maps.LatLng(dealer.lat, dealer.lng);
                    var marker = new google.maps.Marker({
                        position: pos,
                        map: map,
                        title: dealer.name
                    });
                });

                // 用 Handlebars 更新代理商列表
                $('#dealerList').html(dealerTemplate({ dealers: dealers }));

            });

        });
</script>

<div class="dealers">
    <div id="map"></div>
    <div id="dealerList"></div>
</div>
```

注意，因为我们想在客户端用 Handlebars，所以必须用反斜杠转义开始大括号，以防止 Handlebars 试图在后台渲染这个模板。这段代码真正有料的地方是 jQuery 的辅助函数 `.getJSON` 里面（获取 /dealers.json 缓存的地方）。我们在地图上为每个代理商创建一个标记。所有标记都创建完后，用 Handlebars 更新代理商列表。

19.2.4 提升客户端性能

这个简单的例子只适用于有少量代理商的情况。如果要显示上百个标记，或者更多，我们还能从显示中榨出点儿性能。目前我们是解析 JSON 并在其上循环迭代，可以跳过那一步。

我们可以在服务器端直接给出 JavaScript，而不是（或额外）给出代理商的 JSON：

```
function dealersToGoogleMaps(dealers){
    var js = 'function addMarkers(map){\n' +
        'var markers = [];\n' +
        'var Marker = google.maps.Marker;\n' +
        'var LatLng = google.maps.LatLng;\n';
    dealers.forEach(function(d){
        var name = d.name.replace(/'/, '\\\'')
            .replace(/\\/, '\\\\');
        js += 'markers.push(new Marker({\n' +
                '\tposition: new LatLng(' +
```

```
                d.lat + ', ' + d.lng + '),\n' +
            '\tmap: map,\n' +
            '\ttitle: \'' + name + '\',\n' +
          '}));\n';
    });
    js += '}';
    return js;
}
```

然后我们可以把这段 JavaScript 写到一个文件中（比如 /dealers-googleMapMarkers.js），并放在 `<script>` 标签里。地图初始化一完成，我们就可以调用 `addMarkers(map)`，所有标记就都加上了。

这种方式的不足之处在于跟谷歌地图绑定了；如果想切换到必应，只能重新编写服务器端生成 JavaScript 的代码。但如果你需要把速度提到最快，这种办法可行。注意，在产生字符串时必须小心。如果我们简单地用 `"Paddy's Bar and Grill"` 这样的字符串，最终会得到无效的 JavaScript，整个页面都会被毁掉。所以只要遇到字符串，一定要注意你用的字符串分隔符是哪种，并做转义处理。尽管在公司名称中一般不会出现反斜杠，但还是应该确保所有反斜杠都做了转义处理。

19.3　天气数据

还记得第 7 章那个小工具“当前天气”吗？我们来给它挂上真正的数据！用 Weather Underground 的免费 API 获取当地的天气数据。你得创建一个免费账号，在 http://www.wunderground.com/weather/api/ 注册。设置好账号后要创建一个 API 键（得到 API 键后把它放在 credentials.js 文件中，作为 `WeatherUnderground.ApiKey`）。免费 API 有使用限制（在编写本书时每天不能超过 500 次请求，每分钟不能超过 10 次）。为了不超出使用限制，我们会按小时缓存数据。将程序文件中的 `getWeatherData` 函数换成下面这个：

```
var getWeatherData = (function(){
    // 天气缓存
    var c = {
        refreshed: 0,
        refreshing: false,
        updateFrequency: 360000, // 1 小时
        locations: [
            { name: 'Portland' },
            { name: 'Bend' },
            { name: 'Manzanita' },
        ]
    };
    return function() {
        if( !c.refreshing && Date.now() > c.refreshed + c.updateFrequency ){
            c.refreshing = true;
            var promises = [];
            c.locations.forEach(function(loc){
```

```
                var deferred = Q.defer();
                var url = 'http://api.wunderground.com/api/' +
                    credentials.WeatherUnderground.ApiKey +
                    '/conditions/q/OR/' + loc.name + '.json'
                http.get(url, function(res){
                    var body = '';
                    res.on('data', function(chunk){
                        body += chunk;
                    });
                    res.on('end', function(){
                        body = JSON.parse(body);
                        loc.forecastUrl = body.current_observation.forecast_url;
                        loc.iconUrl = body.current_observation.icon_url;
                        loc.weather = body.current_observation.weather;
                        loc.temp = body.current_observation.temperature_string;
                        deferred.resolve();
                    });
                });
                promises.push(deferred);
            });
            Q.all(promises).then(function(){
                c.refreshing = false;
                c.refreshed = Date.now();
            });
        }
        return { locations: c.locations };
    }
})();
// 初始化天气缓存
getWeatherData();
```

如果你不熟悉立即调用函数表达式（IIFE），可能会觉着这个看起来有点儿奇怪。我们基本上是用一个 IIFE 来封装缓存，这样就不会有那么多变量污染全局命名空间。IIFE 返回一个函数，被赋值给了变量 `getWeatherData`，替换了之前返回哑数据那个版本。注意，我们必须再次使用 promise，因为每个地址都要做 HTTP 请求：因为它们是异步的，所以需要用 promise 才能知道什么时候 3 个都完成了。我们还设定了 `c.refreshing`，以防止缓存过期时出现多次、冗余的 API 调用。最后在服务器启动时调用这个函数：如果不调用，则不会填充第一个请求。

在这个例子中，我们把缓存放在了内存中，但我们也完全有理由把这些缓存数据放在数据库里，这样更有利于扩展（允许多个服务器实例访问相同的缓存数据）。

19.4　小结

这真的只是集成第三方 API 所能做的事情中的皮毛。到处都有新的 API 出现，提供各种你能想象得到的数据（波特兰市甚至通过 REST API 开放了很多公众数据）。尽管连可用的一小部分 API 都没可能覆盖到，但这一章已经讲到了使用这些 API 的根本：`http.request`、`https.request` 和解析 JSON。

第20章

调试

“调试”也许是个不幸的词，因为它是跟缺陷相关联的。然而实际上我们所说的“调试”是你一直在做的事情，无论是实现新特性，学习某些东西如何工作，还是真的在解决一个bug。更好的说法可能是“探索”，但我们仍坚持使用“调试”，因为它所指的活动是明确的，不管动机是什么。

调试是一个经常被忽视的技能：看起来好像人们都觉得这是大多数程序员与生俱来的本事。可能计算机科学教授和书籍作者都把调试看成了显而易见的技能，所以忽视了它。

实际上调试是可以传授的技能，并且这是可以让程序员了解他们自己和团队的代码，而不只是他们所用框架的代码的一种重要的方式。

本章会讨论一些工具和技术，你可以用它们有效地调试 Node 和 Express 程序。

20.1　调试的首要原则

从“调试”这个词本身来说，它所指的一般就是找到并去除缺陷的过程。在讨论工具之前，我们先考虑几个通用的调试原则。

> 我跟你说过很多次了，只要把所有不可能的情况都排除掉，不管剩下的可能有多么不可思议，一定是真相。
>
> ——Authur Conan Doyle 爵士

调试中的首要原则是排除的过程。现代计算机系统的复杂性超出人们的想象，如果你必须

把整个系统都放在脑子里，并要从一团乱麻般的线索中找出一个问题的蛛丝马迹，你甚至都不知道如何开始。只要你遇到那种不太显而易见的问题，都应该首先想到："哪些肯定不是这个问题的源头，可以直接排除掉呢？"你能排除的东西越多，所要调查的东西就越少。

排除可能有很多种形态。这里有些常见的例子：

- 系统化地注释掉或禁用代码块。
- 编写能被单元测试覆盖的代码；单元测试本身提供了一个用于排除的框架。
- 分析网络数据流，确定问题是出在客户端还是服务器端。
- 测试系统中跟第一个相似但不同的部分。
- 使用之前能用的输入，并一点一点地修改输入，直到问题呈现。
- 用版本控制逐次回退，直到问题消失。
- "模拟"功能以排除复杂子系统的干扰。

然而排除法也不是高招。问题出现经常是因为两个或更多组件之间复杂的交互：排除（或模拟）其中任何一个组件，问题可能消失了，但并不能肯定问题被隔离在任何一个单独的组件中。然而即便这样，哪怕它无法明确定位到准确的位置，排除也能缩小问题的范围。

只要谨慎小心并方法得当，排除是成功率最高的办法。当你只是想单独排除某些组件，而不考虑这些组件对整体的影响时，很容易漏掉什么。跟你自己玩个游戏：在你考虑要排除某个组件时，完整地考虑一下去掉那个组件会对系统产生什么样的影响。这会让你心里有所准备，不管去掉那个组件能不能给你些有价值的信息。

20.2 利用好REPL和控制台

Node 和浏览器里都有读取 – 计算 – 输出循环（REPL）。这基本上只是一种交互式编写 JavaScript 的方式。你输入一些 JavaScript，按下回车，马上就能看到输出。这是极好的练习办法，一般在定位小段代码中的错误时也是最快、最直观的办法。

在浏览器中，JavaScript 控制台就是你的 REPL。在 Node 中，不带参数运行 `node` 就是进入 REPL 模式；你可以引入包，创建变量和函数，或者做你在代码中能做的任何事（除了创建包，这在 REPL 中没有意义）。

控制台日志也是你的朋友。它可能是一种粗糙的调试技术，但很简单（既易于理解又易于实现）。在 Node 中调用 `console.log` 会以一种易读的格式输出对象中的内容，所以你更容易发现问题。记住，有些对象非常大，把它们输出到控制台中会产生大量的信息，你很难从中找出有用的信息。比如在你的路径处理器中试试 `console.log(req)`。

20.3　利用Node内置的调试器

Node 有个内置的调试器，允许你逐步执行程序，就好像你在用 JavaScript 解释器一样。你只需要在启动程序时加上 debug 参数就可以开始调试了：

```
node debug meadowlark.js
```

在你执行这个命令时，马上就会注意到两件事情。第一，你会注意到控制台里提示调试器在端口 5858 上监听。这是因为 Node 调试器要创建自己的 Web 服务器才能工作，你通过这个 Web 服务器控制被调试程序的执行。这现在对你来说可能没什么，但这种办法的实用性在我们讨论 Node 探查器时就会凸显出来了。

你在控制台调试器中可以输入 help 查看命令列表。最常用的命令是 n（下一步）、s（步入）和 o（步出）。n 会跨“过”当前行：调试器会执行这一行，但如果这条指令调用了其他函数，在这些函数执行完后才会把控制权交回给你。s 与之不同，会进入当前行：如果那一行调用了其他函数，你可以逐步执行它们。o 允许你跳出当前正在执行的函数。（注意，“步入”和“步出”仅指函数；它们不会步入 if 或 for 或其他流程控制语句。）

命令行调试器还有很多功能，但一般你应该不太会去用它。命令行有很多擅长做的事情，但调试不在其列。它的优点是到处都有（比如说，如果你通过 SSH 访问服务器，或者你的服务器根本没装 GUI）。你应该更常用图形界面的调试器，比如 Node 探查器。

20.4　Node探查器

除非是别无他法，否则你可能不会想用命令行调试器，实际上 Node 通过 Web 服务提供了调试控制，所以你还有个可选项。特别是 Danny Coates 的 Node 探查器（现在由 StrongLoop 维护），有了它，你可以在调试客户端 JavaScript 代码的界面里调试 Node 程序。

Node 探查器集成了 Chromium 项目的 Blink 引擎，也就是 Chrome 用的引擎。如果你熟悉 Chrome 的调试器，就会觉得特别自在。如果你完全没做过调试，那么要定下心来开始行动了，第一次实际见到调试器对你还是很有启发的。

很明显，你需要 Chrome（或 Opera；最新版也用 Blink 引擎）。如果你还没有这些浏览器，赶紧去装一个。装好之后安装 Node 探查器：

```
sudo npm install -g node-inspector
```

装好之后需要启动它。如果你愿意，可以在另一个窗口中运行，但除了一个提示性的启动消息，它不会输出太多日志，所以我一般让它在后台运行：

```
node-inspector&
```

> 在 Linux 或 OS X 中，命令最后加一个 & 符号就是让它在后台运行。如果你要把它带回到前台，可以输入 `fg`。如果你一开始运行时没把它放到后台，可以按 Ctrl-Z 让它暂停，然后输入 `bg` 把它切到后台去继续运行。

在启动 Node 探查器时会输出下面这种消息：“访问 http://127.0.0.1:8080/debug?port=5858 开始调试。”除了告诉你如何开始调试，它还告诉你调试界面在 5858 端口（默认端口）。

既然 Node 探查器运行起来了（可以让它运行着……在我的开发服务器上它就是一直运行着的，会自动附着到运行在调试模式的任何程序上），你可以用调试模式启动程序：

```
node --debug meadowlark.js
```

注意，我们用了 `--debug`，而不仅仅是 `debug`；这样你的程序是以调试模式运行的，但不调用命令行调试器（因为用 Node 探查器，所以我们不需要它了）。然后在程序有任何控制台输出之前，你会看到调试器又在监听端口 5858 了，现在一切都就绪了：程序的调试界面在端口 5858 上，Node 探查器在端口 8080 上运行，监听端口 5858。你有三个不同的程序运行在三个不同的端口上！一开始看起来可能觉得眼花缭乱的，但每个服务器都有重要的功能。

我们开始找点乐趣吧：连接 http://localhost:8080/debug?port=5858（记住 localhost 等同于 127.0.0.1）。在浏览器的顶部，你会看到一个带源码和控制台的菜单。如果选择*源码*，你会看到下面有个小箭头。点击那个箭头，你就能看到程序的所有源码。找到主应用程序文件（meadowlark.js）并点击，你就会在浏览器中看到它的源码。

跟我们之前使用命令行调试器的体验不同，程序已经运行起来了：所有的中间件都连进来了，应用正在监听。那么我们如何单步执行代码呢？最简单的办法（可能也是你用得最多的）是设置*断点*。这只是告诉调试器在指定行停止执行，以便你可以单步执行代码。要设置断点，你只需要点击行号（在左边栏中）；上面会出现一个蓝色小箭头，表明这一行上有个断点（再次点击可以关掉）。现在到路由处理器内部设置一个断点。然后在另一个浏览器窗口中访问那个路由。你会发现浏览器卡住了，那是因为调试器注意到了你的断点。

回到调试器窗口，现在单步执行程序的方式比命令行调试器可视化程度更强。设置断点的那行代码是用蓝色高亮显示的。也就是说那是当前执行的那一行代码（实际上是接下来要执行的一行代码）。命令行调试器中用的命令在那里也可以使用。跟命令行调试器类似，我们可以执行以下这些动作。

- *恢复脚本执行*（F8）
  这就是简单的“让它飞”；你不再单步执行代码，直到遇到另一个断点停下来。一般你在看到自己想看的结果后，或者在你想跳到下一个断点时会用到。

- *经过下一个函数调用*（F10）
  如果当前这行代码调用了一个函数，调试器不会进入这个函数中。即这个函数会执行，而调试器会在函数调用完后接着到下一行代码。在遇到你对其细节不感兴趣的函数时可以使用。

- *进入下一个函数调用*（F11）
  这个命令会进入所调用函数的内部，你可以一览无余。如果只用了这个动作，你最终能见到所要执行的一切代码。乍一听比较有趣，但一个小时以后，你会对 Node 和 Express 为你所做的工作有新的发现！

- *步出当前函数*（Shift-F11）
  将会执行你当前所在函数的剩余部分，并在这个函数的调用者中的下一行代码中再次开始调试模式。大多数情况下都是在你不小心进入一个函数，或者已经见到了函数中你所需要见到的东西时，使用这个动作。

除了所有这些控制动作，你还可以访问控制台：这个控制台在你程序的当前上下文中。所以你可以探查变量，甚至修改它们，或者调用函数。这对于实验一些简单的事情来说极其得心应手，但也很容易把你自己搞糊涂，所以我建议你不要过多地用这种方式动态修改运行中的程序；太容易迷糊了。

右侧有一些对你来说有价值的数据。从顶部开始是*观测表达式*，即你可以自己定义的 JavaScript 表达式，可以随着你单步执行程序实时更新。比如说，如果你想跟踪某个特定变量，可以在这里输入它。

观测表达式下面是*调用栈*，你可以从中看出自己是从哪里到达目前这个位置的，即你所在的函数是由哪个函数调用的，而那个函数又是由哪个函数调用的……调用栈把这些函数都列出来了。在 Node 这个高度异步的世界里，要想理清并理解调用栈可能非常困难，特别是涉及匿名函数时。列表中最上面那一行是你当前所在的函数。紧接着下面那行是调用这个函数的函数，以此类推。如果你点击这个列表中的任何一项，就会被神奇地传送到对应的上下文中：你所有的观测和控制台上下文都是在那个上下文中的了。这可能会非常有迷惑性！要真正深入地了解你的程序是如何工作的，这是非常棒的途径，但它不适合那些心脏虚弱的人。因为理清调用栈太难了，所以我在解决问题时把它当作最后一招。

调用栈下边是作用域变量，即目前在作用域中的变量（包括在父作用域中对我们可见的变量）。这一块经常能给你提供你感兴趣的很多关键变量信息，看起来很方便。如果你有很多变量，这个列表会变得臃肿，你最好将你感兴趣的变量定义为观测表达式。

接着是所有断点的列表，真的只是起到记录的作用。如果你在调试有很多细枝末节的问题，并且设了很多断点，有了它还是挺方便的。点击其中一个会把你直接带到那里（但不像在调用栈中那样，它不会改变上下文，这是因为并不是每个断点都一定表示一个活动的上下文，而调用栈中的却一定是）。

最后是 DOM、XHR 和事件监听器断点。这些只适用于在浏览器中运行的 JavaScript，在调试 Node 应用时可以忽略。

有时候你需要调试的是应用程序的设置（比如在你连接中间件到 Express 中去的时候）。像我们之前那样运行调试器，在我们有机会设断点之前，一眨眼的工夫全都发生了。好在这有办法解决。只需要用 `--debug-brk` 代替 `--debug` 就行了：

```
node --debug-brk meadowlark.js
```

调试器会在程序的第一行停住，然后你就可以单步执行，或者设置合适的断点。

要深入了解 Node 探查器（以及一些技巧和提示），请参见项目首页（https://github.com/node-inspector/node-inspector）。

20.5　调试异步函数

当人们第一次接触异步编程时，一般最让人沮丧的就是在调试时。比如看一下下面这段代码：

```
1 console.log('Baa, baa, black sheep,');
2 fs.readFile('yes_sir_yes_sir.txt', function(err, data){
3       console.log('Have you any wool?');
4       console.log(data);
5 });
6 console.log('Three bags full;');
```

如果你刚接触异步编程，可能会觉得自己将会看到：

```
Baa, baa, black sheep,
Have you any wool?
Yes, sir, yes, sir,
Three bags full;
```

但不是。你看到的将是：

```
Baa, baa, black sheep,
Three bags full;
Have you any wool?
Yes, sir, yes, sir,
```

如果你不明白，调试可能也帮不了你。你会从第 1 行开始，单步执行，然后它把你带到第 2 行。然后步入，期望能进入那个函数，最终到第 3 行，但实际上你到了第 5 行！那是因为 `fs.readFile` 只有在它读完函数时才会执行那个函数，而这只有等到程序空闲的时候才会发生。所以你经过第 5 行，来到了第 6 行……你继续试图单步执行，但再也到不了第 3 行了（你最终能到第 3 行，但得等一会儿呢）。

如果你想调试第 3 行或第 4 行，只需要在第 3 行设个断点就可以了，然后让调试器运行。等文件读好了，这个函数被调用时，你就会停在那一行了，希望一切都清楚了。

20.6 调试Express

如果你跟我一样，在职业生涯中见过很多过度设计的框架，单步执行框架源码的想法对你来说可能像是疯了一样（或者是种折磨）。虽然探索 Express 的源码并非儿戏，但对于那些能够理解 JavaScript 和 Node 的人来说也在可掌控的范围之内。并且有时候，当你的代码中有问题时，调试那些问题能通过单步执行 Express 源码本身（或第三方中间件）很好地解决。

本节会简要地介绍一下 Express 源码，以便让你更有效地调试 Express 应用程序。这个介绍中的每一部分都会给出相对于 Express 根路径的文件名（可以在 node_modules/express 下找到），以及函数的名称。我不会具体到行号，因为不同版本的 Express 可能会有不同。

- Express 应用创建（lib/express.js，`function createApplication()`）
  这是 Express 应用生命周期开始的地方。你在代码中声明 `var app = express()` 时，调用的就是这个函数。

- Express 应用初始化（lib/application.js，`app.defaultConfiguration`）
  这是 Express 应用被初始化的地方：这里是观察 Express 所有默认配置的好地方。在这里设置断点几乎毫无必要，但至少应该在这里单步执行一次，可以感受下默认的 Express 设置。

- 添加中间件（lib/application.js，`app.use`）
  每次 Express 链入中间件（不管是你显式执行的，还是由 Express 或其他任何第三方框架显式做的），都会调用这个函数。它看起来很简单，但实际上要真正搞懂还要费一番功夫。有时在这里放个断点很有用（在运行应用时你应该用 `--debug-brk`；否则，你还没来得及设断点所有中间件就已经加进来了），但对你来说可能是很重的负担：你会讶异于一个典型的应用程序中怎么会有那么多中间件加进来。

- 渲染视图（lib/application.js, `app.render`）
  这是另一个相当有料的函数。如果你要调试跟视图相关的刁钻问题，这个函数很有用。如果你单步执行这个函数，会看到 Express 是如何选择和调用视图引擎的。

- 请求扩展（lib/request.js）
  你可能会惊异于这个文件看起来是多么地稀疏简单。Express 添加到 `request` 对象中的方法大多数都是非常简单的便利性函数。因为代码太简单了，所以在上面设置断点或单步执行它们几乎毫无必要。然而，要了解 Express 的便利性方法是如何工作的，看看这个文件一般还是挺有帮助的。

- 发送响应（lib/response.js，`res.send`）
  不管你如何构造响应，`.send`、`.render`、`.json` 或者 `.jsonp`，最后几乎都会到这个函数

上（`.sendFile` 是个例外）。所以在这里设置断点很方便，因为每个响应都会调用它。然后你可以在调用栈中看你是如何到这儿的，对找到问题可能出在哪里非常有帮助。

- 响应扩展（lib/response.js）
  尽管 `res.send` 中有点儿料，但响应对象中的大多数其他方法都非常直白。如果要看你的应用响应请求的确切内容，偶尔可以把断点设在这里。

- 静态中间件（node_modules/serve-static/index.js，`function staticMiddleware`）
  一般而言，如果静态文件没能如期响应，问题应该出在路由上，不会是 `static` 中间件的问题：路由的优先级高于静态中间件。所以如果有一个文件 public/test.jpg 和一个路径 /test.jpg，静态中间件会因为路径而永远不会响应。然而，如果你需要指定如何设置不同静态文件的响应头，单步执行静态中间件可能有用。

如果你抓破头也想不出这些中间件都在哪儿，那是因为 Express 很少有中间件（静态中间件和路由器显然是例外）。大多数中间件实际上是来自于 Connect 的，我们接下来会讨论到。

因为 Express 4.0 不再捆绑 Connect 了，你要单独安装 Connect，所以你能在 node_modules/connect 中找到 Connect 的源码（包括它的所有中间件）。Connect 也把它的一些中间件剥离到独立的包里去了。下面是一些比较重要的包的位置。

- 会话中间件（node_modules/express-session/index.js，`function session`）
  实现会话要做很多工作，但代码相当直白。如果你遇到跟会话有关的问题，可能想要在这个函数里设置断点。记住，为会话中间件提供存储引擎是你的责任。

- 日志中间件（node_modules/morgan/index.js，`function logger`）
  日志中间件真的是为了帮你调试而存在的，不是为了让你调试它。然而，日志工作有些微妙的地方，你可能偶尔要单步执行它。我第一次这么干的时候，有很多让我感叹的地方，并且后来使用日志的效果更好了，所以我建议你至少浏览它一遍。

- URL 编码请求体解析（node_modules/body-parser/index.js，`function urlencoded`）
  请求体的解析方式对人们来说经常是一个谜。实际上并没有那么复杂，单步执行这个中间件有助于你理解 HTTP 请求的工作方式。除了学习体验，你应该不会经常因为调试而到这个中间件里来。

我们在本书中讨论了很多中间件。我无法把你在 Express 内部之旅中的每个地标都列出来，但希望这些重点能帮你去除一些 Express 的神秘感，并让你在有需要时敢于探索这个框架的源码。中间件不仅在品质上有很大差别，在可访问性上也是如此：有些中间件的难以理解程度堪称邪恶，而有些又清澈如水。不管怎样，要勇于探索：如果它太难，你可以走开（当然，除非你真的需要理解它）；如果不是那么难，你可能会学到一些东西。

第21章

正式启用

大日子终于来了：经过几周甚至几个月的辛苦劳作，你的网站或服务已经准备就绪，可以启用了。但激活网站并不像扳一下开关那么容易，或者是那么容易吗？

本章（你真的应该在正式启用前几周看，别等到那天才看！）将会介绍一些域名注册和托管服务，从临时环境向生产环境迁移的技术，部署技术，以及选择生产服务时应该考虑的事情。

21.1 域名注册和托管服务

人们经常搞不清楚*域名注册*和*托管服务*之间的区别。作为本书的读者，你可能不在那些人之列，但我打赌你知道那些人是谁，比如你的客户或你的经理。

互联网上的所有网站和服务都可以由一个互联网协议（IP）地址标识（或者不止一个）。人们不太容易记住这些数字（随着 IPv6 的普及，这种状况会愈演愈烈），但计算机最终需要这些数字来显示网页。所以就有了域名。它们将一个人类容易记住的名字（比如 google.com）和一个 IP 地址（74.125.239.13）映射起来。

如果要用现实世界的事物类比，它们之间的差别就好像公司名称和物理地址一样。域名就像公司名称（苹果），而 IP 地址就像物理地址（1 Infinite Loop, Cupertino, CA 95014）。如果你要开车去苹果总部，需要知道物理地址。好在如果你知道公司名称，很可能也能找到物理地址。采用这种抽象的另一个原因是这有助于组织迁移（搬到一个新的物理地址去），即便它搬家了，人们仍然可以找到它。

而托管服务器描述的是运行网站的真实计算机。我们继续用类比来解释这个概念，托管服务器相当于你到达物理地址时见到的真实建筑。人们经常搞不清楚域名注册和托管服务器的关系其实非常小，并且你几乎不会从提供托管服务的商家那里购买域名（就好像你一般是跟一个人买地，再付钱给另一个人帮你盖房子并维护它们）。

尽管没有域名也可以托管你的网站，但那样非常不友好：IP 地址很不好推广！一般来说，当你购买托管服务时，会自动分配一个子域名（马上就会讲到），可以看作是易于推广的域名和 IP 地址之间的东西（比如 ec2-54-201-235-192.us-west-2.compute.amazonaws.com）。

一旦有了域名，并且正式上线后，你可以通过多个 URL 访问网站。比如：

- http://meadowlarktravel.com/
- http://www.meadowlarktravel.com/
- http://ec2-54-201-235-192.us-west-2.compute.amazonaws.com/
- http://54.201.235.192/

由于域名映射，这些地址全都指向同一个网站。请求到达网站时，基于所用 URL 采取动作是有可能的。比如说，如果有人通过 IP 地址访问网站，你可以自动重定向到域名上，尽管这种情况不太常见，因为几乎没有指向它的（更常见的是从 http://meadowlarktravel.com/ 转向 http://www.meadowlarktravel.com/）。

大多数域名注册机构都提供托管服务（或者跟托管公司合作）。我从没见过哪个注册机构能提供特别有吸引力的托管服务，我建议将域名注册和托管分开，这样更安全，也更灵活。

21.1.1　域名系统

域名系统（DNS）负责将域名映射到 IP 地址。这个系统相当复杂，但作为站长，有些跟 DNS 有关的知识你应该掌握。

21.1.2　安全

你应该时刻牢记域名的价值。如果你的托管服务被黑客完全攻破，托管主机被控制了，但只要你还能控制自己的域名，就可以找一台新的托管主机，把域名转过去。换句话说，如果你的域名被攻破了，那就真麻烦了。

你的名声是跟域名绑在一起的，并且好域名都要非常认真地保护好。那些无法控制自己域名的人将会发现自己遭受到的打击是毁灭性的，并且世界上有些人热衷于窃取域名（特别是那种特别短或特别好记的），他们可以把它卖掉，或者毁坏你的名声，或者勒索你。因此你应该非常严肃地对待域名的安全问题，它甚至比数据安全更重要（取决于你的数据有

多重要)。我曾见过一些人，他们在主机安全上花了过多的时间和金钱，却找那种最便宜、最劣质的域名注册服务。不要犯这样的错误。(好在高品质的域名注册不是特别贵。)

考虑到保护域名注册所有权的重要性，你应该采用跟域名注册相称的安全实践。最起码应该使用具有唯一性的强密码，并且采用正确的密码管理策略（别把它写在便条上，然后贴显示器上)。你最好用那种提供双重身份认证的注册商。不要害怕向注册商提出修改账号授权需要什么这种尖锐问题。我推荐两个域名注册商：Name.com 和 Namecheap.com。这两家都用了双重身份认证，并且我发现他们的客户服务做得很好，在线控制面板既简单又健壮。

在注册域名时，你必须提供一个跟域名关联的第三方邮件地址（比如说，如果注册 meadowlarktravel.com，则应该用 admin@meadowlarktravel.com 作为注册邮箱)。因为任何安全系统的强度取决于它最弱的环节，所以应该用一个安全性强的邮件地址。比较常用的是 Gmail 或 Outlook 账号，并且如果你这样做了，那么邮件地址所用的安全标准应该和域名注册账号一样（良好的密码管理策略和双重身份认证)。

21.1.3 顶级域名

域名的结尾部分（比如 .com 或 .net）叫作顶级域名（TLD)。一般来说，有两类 TLD：国家代码 TLD 和通用 TLD。国家代码 TLD（比如 .us、.es 和 .uk）是用来提供地理区域分类的。然而谁能获取这些 TLD 有一些限制（毕竟互联网是个全球性网络)，因此它们经常用于“机灵的”域名，比如 placehold.it 和 goo.gl。

通用 TLD（gTLD）包括大家熟悉的 .com、.net、.gov、.fed、.mil 和 .edu。所有人都可以获取可用的 .com 或 .net 域名，但刚才提到的其他域名却有申请限制。更多信息请参阅表 21-1。

表21-1　受限的gTLD

TLD	更多信息
.gov、.fed	https://www.dotgov.gov
.edu	http://net.educause.edu/edudomain
.mil	军事人员和承包商应该联系他们的 IT 部门，或者美国国防部网络信息中心（http://www.disa.mil/Services/Network-Services/Service-Support）

顶级域名的管理是由互联网名称与数字地址分配机构（ICANN）最终负责的，不过他们把大部分实际管理工作交给其他组织代理。ICANN 最近授权了很多新的 gTLD，比如 .agency、.florist、.recipes，甚至 .ninja。在可以预计的未来，.com 可能仍将作为“优质”TLD，并且是最难取得的资产。那些在互联网成长期购买到了优质 .com 域名的人非常幸运（或精明)，得到了丰厚的回报（比如 Facebook 在 2010 年以高达 8 500 000 美元的

价格买下了 fb.com）。

因为 .com 域名比较稀缺，人们转向了其他 TLD，或者用 .com.us 来更加准确地反映他们的组织。在挑选域名时，你应该考虑将会如何使用它。如果你计划主要通过电子媒体（那里的人们更喜欢点击链接而不是输入域名）推广它，那你应该致力于获取一个引人注目的或有意义的域名，而不是短域名。如果你准备集中力量做平面广告，或者有理由相信人们会在自己的设备中手动输入你的 URL，你可能要考虑其他的 TLD，以便得到更短的域名。拥有两个域名的情况也很常见：一个短的，易于输入的域名；一个较长，适合推广的域名。

21.1.4　子域名

TLD 在域名后面，子域名在域名前面。目前最常见的子域名是 www。对这个子域名我从来没特别关心过。毕竟你是在计算机上，用着万维网；我非常肯定，即便没有 www 的提醒，人们也不会糊涂。因此我建议主域名别用子域名：用 http://meadowlarktravel.com/ 代替 http://www.meadowlarktravel.com/。它更短更轻松，并且因为有重定向机制，也不用担心那些习惯于用 www 开头输入网址的用户访问不到你的网站。

子域名也用于其他用途。像 blogs.meadowlarktravel.com、api.meadowlarktravel.com 和 m.meadowlarktravel.com（用于移动站点）之类的网址很常见。一般这样做是出于技术上的原因，比如说，如果你的博客用的服务器跟网站其他部分完全不同，用子域名更容易。然而一个好的代理既能根据子域名重定向流量，也能根据路径重定向，因此选择使用子域名还是路径应该更侧重于内容而不是技术（Tim Berners-Lee 说过，URL 表示的是信息架构，而不是技术架构）。

我建议用子域名给网站或服务有显著区别的部分分区。比如说，我认为用 api.meadowlarktravel.com 提供 API 是子域名的良好用法。微站（跟网站其余部分外观不同的站点，通常是为了突出某个产品或主题）也应该用子域名。子域名的另一个明智用途是将管理界面跟公众界面分开（admin.meadowlarktravel.com，只供员工使用）。

除非特别指明，否则域名注册商会忽略子域名将所有请求都重定向到你的服务器。然后由服务器（或代理）根据子域名采取相应的动作。

21.1.5　域名服务器

让域名生效的“胶水”是域名服务器，在搭建网站的服务器时需要提供。一般这个相当简单，因为你的托管主机服务提供商会帮你做好大部分工作。比如说，我们选择把 meadowlarktravel.com 放在 WebFaction（http://www.webfaction.com）的主机上。当你设置 WebFaction 的托管主机账号时，WebFaction 会给你域名服务器的名称（为了冗余会有多个）。WebFaction 跟大多数托管主机提供商一样，管他们的域名服务器叫 ns1.webfaction.

com、ns2.webfaction.com，等等。到域名注册商那里给你要托管的域名设好域名服务器就可以了。

在这个例子中，映射的工作方式是：

(1) 用户访问网址 http://meadowlarktravel.com/。
(2) 浏览器发送请求到用户计算机的网络系统。
(3) 用户计算机的网络系统中有网络接入商给出的互联网 IP 地址和 DNS 服务器，会要求 DNS 服务器解析 meadowlarktravel.com。
(4) DNS 服务器知道 meadowlarktravel.com 是由 ns1.webfaction.com 处理的，所以要求 ns1.webfaction.com 给出 meadowlarktravel.com 的 IP 地址。
(5) 服务器 ns1.webfaction.com 收到请求，认出 meadowlarktravel.com 确实是活跃账号，返回与之关联的 IP 地址。

尽管这是最常见的情况，但并不是配置域名映射的唯一方式。既然真正提供网站服务的服务器（或代理）有 IP 地址，我们可以将那个 IP 地址直接注册到 DNS 服务器上，从而砍掉中间环节（这可以有效去掉前面那个例子中的域名服务器 ns1.webfaction.com）。要使用这种方式，你的托管主机必须有一个静态 IP 地址。托管服务提供商一般会给你的服务器分配一个动态 IP，即这个 IP 可能会未经通知直接变动，这样这种方式就是无效的。有时静态 IP 要额外付费：跟你的托管服务提供商核实一下。

如果你想把域名直接映射到网站上（跳过域名服务器），你可以添加一个 A 记录或 CNAME 记录。A 记录将域名直接映射到一个 IP 地址，而 CNAME 将域名映射到另一个域名上。CNAME 记录通常缺乏一点儿灵活性，所以 A 记录一般更受欢迎。

不管用哪种技术，域名映射一般是积极缓存的，也就是说如果你修改了域名记录，可能需要 48 小时才能把你的域名对应到新服务器上。记住，这也和地理位置有关系：即便你看到域名在洛杉矶能用了，在纽约的客户访问到的域名可能还是指向之前那台服务器的。按我的经验，在美国本土内一般只需要 24 小时域名就可以正确解析了，而在国际上需要 48 小时。

如果你需要在确定的时间准确启用，则不应该依赖 DNS 改动。而是应该让你的服务器转到“马上推出”站点或页面，然后在真正切换之前预先修改 DNS。这样在预定时刻，你可以将服务器切到正式启用的站点，这样当用户访问时，不管他们在世界上的什么地方，都能马上见到变化。

21.1.6　托管

选择托管服务乍一看可能很困难。Node 已经取得了长足的发展，所有人都宣传能提供 Node 托管服务以满足这种需求。如何选择托管服务提供商在很大程度上取决于你的需求。

如果你有理由相信自己的网站会是下一个亚马逊或 Twitter，那你所关心的问题跟那种为本地集邮俱乐部构建的网站所关心的问题有很大差异。

1. 传统托管，还是云托管

“云”是最近几年突然出现的含义最模糊的技术术语之一。真的，它只是用一种很炫的方式说“互联网”，或者“互联网的一部分”。不过这个术语也不是毫无用处。尽管这个术语中没有技术定义部分，托管在云中一般表示计算资源在一定程度上的商品化。也就是说，我们不再把“服务器”当作一个独立的物理实体：它只是在云中某处的一个同质化资源，它们都一样好。当然，我过于简化了：计算资源是按照内存、CPU 数量（以及定价）等加以区分的。就你所要了解（和关心）的角度而言，把应用部署在真正的服务器上和把它部署在云中的服务器上两者之间的区别是，应用能在你不知情（或关心）的情况下轻松迁移到不同的云服务器上。

云托管是高度虚拟化的。也就是说运行应用的服务器一般不是真实的物理主机，而是运行在真实服务器上的虚拟机。这个概念并不是云托管引入的，但云托管已经变成了它的代名词。

尽管云托管并不是什么新东西，但它确实代表了认识上的微妙变化。这个概念一开始可能令人有点儿不安，对你的服务器所在的真实物理机器毫不知情，相信你的服务器不会被运行在同一台机器上的服务器影响。然而真的什么都没变：当你的托管账单过来时，你本质上还是为相同的东西付钱：有人照顾那些让你的 Web 应用程序跑起来的物理硬件和网络。唯一改变的只是你离硬件更远了。

我相信“传统”托管（没有更好的词）最终会消失。但那不是说托管公司会倒闭（尽管有些终究会），他们只是开始提供云托管。

2. XaaS

在考察云托管时，你会遇到 SaaS、PaaS、IaaS 这几个缩写。

- 软件即服务（SaaS）
  SaaS 一般用来描述提供给你的软件（网站、应用）：你只是使用它们。谷歌文档和 Dropbox 就是这样的软件。

- 平台即服务（PaaS）
  PaaS 为你提供了所有的基础设施（操作系统、网络，所有都弄好了）。你只需要编写应用程序。尽管 PaaS 和 IaaS 之间的界限比较模糊（作为开发者，你会发现自己经常会跨过这条线），这一般是我们在本书中讨论的服务模型。如果你运营着一个网站或网络服务，PaaS 可能就是你要找的东西。

- 架构即服务（IaaS）
  IaaS 最灵活，但也是有代价的。它只是提供虚拟机和基本的网络连接。然后你负责安装

和维护操作系统、数据库和网络策略。除非你需要对环境做这种层面的控制，否则一般还是会选 PaaS。（注意，PaaS 确实允许你控制操作系统和网络配置的选择，只是你不必亲自动手实现。）

3. 大型托管

随着计算资源的商品化，那些基本上掌控着互联网（或者至少是在互联网的运行上有巨大投入）的公司意识到他们还有个产品可以卖。微软、亚马逊和谷歌全都提供云计算服务器，并且他们的服务都挺好的。

这些服务的价格都差不多：如果你的要求不高，这三家的价格几乎没什么差别。如果你需要很高的带宽或存储要求，那就要好好研究一下，因为根据你的需求，成本上可能会有很大差异。

尽管在考虑开源平台时我们一般不会想到微软，但我不会忽略 Azure。不仅因为它是一个成熟健壮的平台，还因为微软已经放下了身段，Azure 不仅对 Node 友好，对开源社区也很友好。Azure 提供了一个月的试用期，你可以借此确定它的服务能否满足你的需求；如果你考虑在这三大服务商中间选一个，我强烈推荐你试用一下 Azure 的免费服务，对它进行评估。微软为他们所有主要服务提供了 Node API，包括云存储服务。除了优秀的 Node 主机，Azure 还提供基于 Git 的部署，一个优秀的云存储系统（有 JavaScript API），以及良好的 MongoDB 支持。Azure 的不足之处是他们没有为小项目提供定价层。用 Azure 的生产型主机一个月最少要付 80 美元。但这个价格让你可以轻松部署多个项目，所以如果你希望整合一堆网站，它的性价比还是很高的。

亚马逊提供了最完备的资源组合，包括 SMS（短信）、云存储、邮件服务、支付服务（电商）、DNS 等。此外，亚马逊还提供免费试用层，非常易于评估。

谷歌的云平台还没有为 Node 托管提供服务，但 Node 应用可以托管在他们的 IaaS 服务上。谷歌目前不提供免费层或服务试用。

除了“三大”，Joyent 也值得考虑，它目前在 Node 开发中参与程度很高。Joyent 的合作伙伴 Nodejitsu 提供了专门针对 Node 的托管服务和领域专家。他们为开发提供了独一无二的选择：私有 npm 存储库。如果你不喜欢基于 Git 的部署（我们会在本书中专门讨论），我建议你研究下 Nodejitsu 基于 npm 的部署。

4. 精品托管

比较小型的托管服务，我准备称之为“精品”托管服务（没有比较好的词），可能没有微软、亚马逊或谷歌那样的基础设施或资源，但并不是说他们不能提供有价值的东西。

因为精品托管服务不能在基础设施上跟人竞争，所以他们一般更加重视客户服务和支持。如果你需要大量支持，可能要考虑精品托管服务。对于个人项目而言，我用 WebFaction

（http://webfaction.com）很多年了。他们的服务极其实惠，并且他们已经提供 Node 托管有段时间了。如果你有过之前合作愉快的托管服务提供商，别犹豫，问问他们是否提供（或计划提供）Node 托管。

21.1.7　部署

在 2014 年还有人用 FTP 部署应用程序，这太让我诧异了。如果你也是那样，请停下来吧。FTP 绝对不安全。不仅你的所有文件传输都是未经加密的，连用户名和密码的传输也是未经加密的。如果你的托管服务提供商只提供了这一种方式，再找一家吧。如果你确实没有其他选择，一定确保你所用的密码不用在其他地方。

最起码你也应该用 SFTP 或 FTPS（别搞混了），但还有更好的办法：基于 Git 的部署。

这个想法很简单：不管怎样，你都会用 Git 做版本控制，并且 Git 在版本控制上做得非常出色，而部署本质上也是个版本问题，所以 Git 是很自然的选择。（这项技术不仅限于 Git；你也可以用 Mercurial 或 Subversion 部署。）

要使用这个技术，需要想办法把你的开发存储库和部署存储库同步起来。Git 为此提供了几乎数不清的办法，但目前最容易的是用 GitHub 这样的互联网服务。GitHub 的公开存储库是免费的，但你可能不想把网站的源码公开。可以付费升级到私有 Git 存储库。此外，Atlassian Bitbucket 提供了五个用户的免费私有存储库。

尽管基于 Git 的部署可以设置在几乎所有的服务上，但 Azure 提供的服务是开箱即用的，并且他们的实现很棒，实现了基于 Git 部署的承诺。我们会从这个优秀的模型开始，然后介绍如何在其他托管服务提供商上部分模拟这一模型。

1. Git部署

Git 最强的是它的灵活性（也是最大的弱点）。它几乎可以适应你能想到的任何工作流。为了部署，我建议创建一个或多个专门针对部署的分支。比如说，你可能有一个 `production` 分支和一个 `staging` 分支。如何使用这些分支完全取决于你自己的工作流。一种比较流行的方式是从 `master` 到 `staging` 再到 `production`。所以一旦 `master` 上的某些修改可以启用了，你就可以把它们合并到 `staging` 中。一旦它们在临时服务器上得以证实可用，你就可以把它们合并到 `production` 中。尽管这个在逻辑上讲得通，但我不喜欢这么繁琐（到处合并）。还有，如果你有很多功能要放到临时区并以不同的顺序推送到生产区，很快就会搞得一团糟。我觉得更好的方式是把 `master` 合并到 `staging`，然后当你准备好启用这些修改时，把 `master` 合并到 `production` 中。这样 `staging` 和 `production` 的关联更少了：你甚至可以多开几个 `staging` 分支来试验不同的功能，然后再正式启用（并且你还可以把 masert 之外的东西合并进去）。只有那些被证实可以放到生产环境中时，再把它们合并进 `production`。

当需要回滚变化时会怎么样？这里可能会变得比较复杂。有几种技术可以取消修改，比如应用逆向提交来取消前面的提交（`git revert`），但这些技术不仅仅是复杂，可能还会引发后续的问题。我建议将 `production`（如果你愿意的话，包括 `staging` 分支）当作一次性的：它们实际上只是你的 `master` 分支在不同时点的映像。如果你需要回滚修改，只需要在你的 `production` 分支上做一次 `git reset --hard <old commit id>`，然后 `git push origin production --force`。这在本质上是“重写历史”，经常被教条式的 Git 使用者描述为危险的或“高级的”行为。然而在这里绝对可以理解为 `production` 是一个只读分支；开发人员决不能向它提交代码（重写历史会给你带来麻烦）。

最后，Git 的工作流是由你和你的团队决定的。更重要的是你们选择的工作流跟你们使用它，以及围绕它开展的训练和沟通是一致的。

> 我们已经讨论过把二进制资产（多媒体和文档）跟代码库分开的价值了。基于 Git 的部署为这种方式提供了另一个动力。如果你的存储库中有 4G 的多媒体数据，要克隆它们需要花很长时间，并且你的每个生产服务器上都有一份没必要存在的数据副本。

2. 部署到Azure

在 Azure 上，你可以从 GitHub 或 Bitbucket 存储库上部署，也可以从本地存储库部署。我强烈推荐你使用 GitHub 或 Bitbucket，这样在往开发团队里加人时会更容易。在后续的例子中，我们或者用 GitHub，或者用 Bitbucket（两者的过程几乎相同）。你需要在 GitHub 或 Bitbucket 账号下设置一个存储库。

有一点必须提一下，Azure 希望你的主应用程序文件名是 server.js。我们之前用的是 meadowlarktravel.js，所以如果要部署到 Azure 上，必须把它改成 server.js。

登录到 Azure 主界面之后，你可以创建一个新网站：

(1) 点击左侧的 Website 图标。
(2) 点击底部的 New。
(3) 选择 Quick Create。选择名称和区域，并点击 Create Web Site。

然后设置源码控制部署：

(1) 在主界面窗口上点击你的网站。
(2) 在“Your site has been created!”消息下面，找到“Set up deployment from source control”。点击那个链接。
(3) 选择 GitHub 或 Bitbucket。如果这是你的第一次，Azure 会要求你授权访问你的 GitHub 或 Bitbucket 账号。
(4) 选择你要用的存储库和分支（我建议用 `production`）。

这就是你要做的所有工作，现在神奇的事情发生了。如果 Azure 检测到 `production` 分支有更新，它会自动更新服务器上的代码（我已经这样做了几百次了，从来没有一次超出过 30 秒，不过如果你做了非常大的改动，比如多媒体资产，可能要花更长的时间）。甚至更好？如果你往 package.json 中添加了任何新的依赖项，Azure 会自动替你安装。它还会处理文件检测（不要吃惊，因为这是 Git 的标准行为）。换句话说，你这是无缝开发。

基于 Git 的部署不仅仅是无缝，如果你要扩展程序，这项技术也很好用。所以如果你有四个运行的服务器实例，只要推到合适的分支上，就把所有服务器都同时更新了。

如果你访问 Azure 给你网站的控制面板，会看到一个标题为部署的标签页。在这个标签页中有部署历史的信息，在你的自动部署系统出问题时可能有助于调试。还有，你可以重新部署之前部署的版本，如果有问题，还能快速恢复原状。

3. 基于Git的手工部署

如果你的托管服务提供商不支持基于 Git 的任何类型的自动化部署，你还要再做些工作。比如说你的设置是一样的：用 GitHub 或 Bitbucket 做版本控制，有一个 `production` 分支，要反应到生产服务器上。

你必须为每个服务器克隆存储库，检出 `production` 分支，并且设置好启动 / 重启程序所必需的基础设施（这要看你选的什么平台）。当你更新 `production` 分支时，必须到每台服务器上运行 `git pull --ff-only`，运行 `npm install`（如果你更新过依赖项），然后重启程序。如果你的部署不是很频繁，并且服务器不多，这可能不是什么大问题，但如果你要频繁更新，很快你就会受不了，希望能找到自动化的实现方式。

> `git pull` 的 `--ff-only` 参数只允许快进 pull，防止自动合并或重订。如果你知道 pull 是只快进的，可以忽略它，但如果你习惯那么做，绝不会不小心调用了合并或重订！

可惜自动化不是那么简单的。Git 有让你执行自动化动作的钩子，但前提是更新的不是远程存储库。如果你要实现自动部署，最容易的方式是运行一个自动化任务，定期执行 `git pull --ff-only`。如果有更新，再运行 `npm install` 并重启应用。

4. 在亚马逊上用Elastic Beanstalk部署

如果你在用亚马逊的 AWS，则可以用他们的 Elastic Beanstalk（EB）实现 Git 自动部署。EB 是个复杂的产品，提供了很多功能，如果你在部署中绝对不能犯错，会觉得它非常有吸引力。然而随着这些功能变得越来越复杂，用 EB 设置自动部署相当复杂。在 EB 文档页（http://aws.amazon.com/cn/elasticbeanstalk/）上有各种配置 EB 的办法。

21.2 小结

部署网站（特别是第一次）应该是一个激动人心的时刻。应该有香槟和欢呼声，但还有那么多的汗水、咒骂和熬夜。我见过太多烦躁的、筋疲力尽的团队在凌晨 3 点推出的网站。还好，这种状况正在改观，这部分归功于云部署。不管你选择什么部署策略，你能做的最重要的事情是尽早开始生产部署，而且是在网站可以正式启用之前。你没必要把域名挂上，所以公众没必要知道。如果你在正式推出之前已经往生产服务器上部署过很多次了，成功推出的机会要高很多。 理想情况下，你的网站在推出之前已经在生产服务器上运行很久了，你所要做的只是把老站点切换到新站点上。

第22章

维护

你把网站推出去了！祝贺你，现在你可以把它抛到脑后了。什么？事情还没完？好吧，如果是这样，请继续往下看。

在我的职业生涯中，只有那么一两次，网站做完后我就再也没碰过它（即便如此，一般也是因为有别人做这些事，并不是不用做这个工作）。我清楚地记得有一次网站推出时，同事们管这叫“验尸”（postmortem）。我插嘴问道：“难道我们不应该管它叫‘产后’（postpartum）吗？”[1] 推出一个网站更像是一个生命的诞生而不是死亡。一旦推出，你就会被分析工作绑住，焦虑地等着客户的反应，凌晨 3 点起床去检查网站是不是还在运行……它就像你的宝宝一样。

圈定网站的范围、设计网站、搭建网站，这些都是能规划到死的活动。但网站的维护在规划时一般会受到冷遇。本章会就此给你一些建议。

22.1　维护的原则

22.1.1　有长远规划

有些客户在同意搭建网站所需的费用时从不说明期望这个网站持续多长时间，这时候我总会比较吃惊。按我的经验，如果工作做得好，客户付钱时也会很开心。但客户不喜欢意外：3 年后你告诉他们网站需要重建，而客户本来期望网站可以坚持 5 年（不过却没有明说）。

注 1：那时候说“验尸”觉得有点儿太过了。现在我们管它叫“回顾”（retrospective）。

互联网发展得很快。即便你构建网站时用的绝对是你能找到的最好和最新的技术，但在短短的两年后也会觉得像个摇摇欲坠的古董了。或者它能延续 7 年之久，虽然老迈，但还能很优雅地运行（这种情况很少见！）。

设定对网站的长期期望混杂着艺术、销售技巧和科学的成分在里面。科学的成分都是科学家做的，开发人员很少做：保存记录。想象一下，你有你们团队推出的所有网站的记录，维护请求和失败的历史，用过的技术，以及每个网站在重做之前用了多长时间。很明显，这里有很多变量，从参与的团队成员到经济因素，再到技术风向的变化，但那不意味着在这些数据中挖掘不出有意义的趋势。你可能会发现某种开发方式更适合你们的团队，或者某个平台，某项技术。我几乎可以向你保证，一定能在“拖延”和缺陷之间发现相关性：你在基础设施更新或平台升级上拖得越久，情况就越糟糕。有一个良好的问题追踪系统，并一丝不苟地把记录保存下来，客户会对项目的生命周期如何演进有更好（也更切合实际）的认识。

推销成分当然可以归结到钱。如果客户完全有能力每隔三年完全重建他们的网站，那么他们可能不太愿意遭受基础设施老化之苦（尽管他们还会遇到其他问题）。另一方面，也有些客户要让他们的钱充分发挥作用，想让网站持续 5 或 7 年（我知道有些网站甚至坚持了更长时间，但我觉得 7 年是一个网站可能还能发挥作用的最长年限）。你对这两种客户都负有责任，对于那些有很多钱的客户，不要因为他们有钱就拿他们的钱：用额外的钱给他们一些超值的东西。对于预算紧张的客户，你必须以创造性的方式设计他们的网站，让它在不断变化的技术中能够持续更长时间。这两个极端都有它们的困难之处，但可以解决。重要的是你知道期望是什么。

最后是艺术的成分。是它把一切融合在一起：理解客户能承担什么，你能在哪里真诚地说服客户，让他们花更多钱得到他们需要的价值。它也是理解技术未来发展趋势的艺术，并且能够预测什么技术会在 5 年内被痛苦地淘汰，什么技术会变强。

当然，做出绝对准确的预测是不可能的。你可能赌错技术，人员变动可能完全改变组织的技术文化，技术供应商可能会破产（尽管在开源世界中一般不太可能出现这种问题）。你认为在产品整个生命周期中都能坚如磐石的技术可能最终被证明只是一阵潮流，你会发现自己不得不比预期更快地做出重建决定。另一方面，有时是恰当的团队在恰当的时间带着恰当的技术走到了一起，所创造的东西持续的时间也超出了任何合理的预期。然而所有这些不确定性都不应该阻扰你确立一个规划：有个出错的计划总好过总是毫无章法。

现在你应该清楚了，我觉得 JavaScript 和 Node 还要持续一段时间。Node 社区充满活力，激情四射，明智地选择了一个明显胜出的语言。也许最重要的是 JavaScript 是一个多范式语言：面向对象、函数式、过程化、同步、异步——它全都有。因此 JavaScript 对各种不同背景的开发人员都很包容，并且在很大程度上对 JavaScript 生态系统中革新的节奏负责。

22.1.2　使用源码控制系统

这对你来说可能是显而易见的，但它不止是使用源码控制系统，还要用好。你为什么要用源码控制系统？理解原因，并确保工具支持那些原因。用源码控制系统有很多原因，但对我来说最大的收益还是它的根本属性：知道什么时候谁做过什么修改，这样在有必要时我就可以询问更多信息。要了解项目的历史，以及我们是如何到一起成为一个团队的，版本控制是最好的工具之一。

22.1.3　使用问题追踪系统

问题追踪系统又回到了开发科学上。没有系统化的项目历史记录办法，是不可能对项目有什么深刻认识的。你可能听说过，所谓疯狂就是“一次次做同样的事情却期望得到不同的结果”（经常有人说这是爱因斯坦说的）。一次次重复自己的错误看起来确实疯狂，但如果你不知道自己犯过什么错误，又怎么能避免这种情况出现呢？把所有事情都记下来：客户报告的每条缺陷；在客户发现之前被你找出的每条缺陷；每次抱怨，每个问题，每一点称赞。记录它用了多长时间，谁修订的，涉及哪些 Git 提交，谁确认了修订。这里的艺术是找到合适的工具，别让这个工作太耗时间或太繁重。糟糕的问题追踪系统会受到冷遇，没有人用，并且会变得比毫无用处还糟糕。好的问题追踪系统能让你对业务、团队和客户有更深刻的认识。

22.1.4　良好的卫生习惯

我不是说你要刷牙——尽管你也确实应该刷牙——我说的是版本控制、测试、代码审查和问题追踪。只有你真的在用，并且用得正确，工具才真的有用。代码评审是鼓励卫生习惯的好方式，因为所有东西都会触及到，从发出请求中讨论问题追踪系统的使用，到必须添加测试来证实修订，到针对版本控制提交的评论。

应该定期评审从问题追踪系统中收集的数据，并跟团队讨论。从这些数据中，你可以得到什么有用、什么没用的深刻认识。你可能会对自己的发现感到吃惊。

22.1.5　不要拖延

机构拖延症是最难以战胜的困难之一。一般它看起来没那么糟：你注意到可以通过一次小重构极大缩减团队在每周更新上花掉的大量时间。你每周推迟的重构都会让你在另一周付出效率低下的代价。[1] 更糟的是有些代价会随着时间增长。不去更新软件依赖项就是很好的例证。随着软件变老，团队成员的更替，越来越难找到还记得（或者曾经明白）这个老软件的人。支持社区开始变得薄弱，并且不久所用技术也被废止了，再找不到任何支持。你

注 1：Mike Wilson of Fuel 的经验法则是：“如果你是第三次做某件事了，那就花时间让它自动化。”

经常听到有人把这个叫作技术债务，而它真的会发生。尽管你应该避免拖延，但同时也要明白网站的长寿可能会导致下面这种决策：如果你正要重新设计整个网站，那就没必要去消除积累起来的技术债务。

22.1.6　做常规的QA检查

对于你的每个网站，都应该有落实到文档上的常规 QA 检查。这些检查应该包括链接检查、HTML 和 CSS 校验和运行测试。这里的关键是文档：如果组成 QA 检查的项目没有记录在文档中，则你终将会漏掉什么。每个网站一个文档化的 QA 检查列表不仅能防止人们忽视检查，还可以让新的团队成员迅速上手。理想情况下，QA 检查列表能由非技术人员执行。这样不仅能让团队中的非技术管理者（可能）更有信心，如果你没有专职的 QA 部门，这样还能将 QA 职责分散开。根据你和客户的关系，你可能还想跟客户分享 QA 检查列表（或者其中的一部分）；这样能提醒他们自己在为什么付钱，并且你在寻找他们的最佳利益。

我推荐你在常规 QA 检查中使用谷歌站长工具（https://www.google.com/webmasters）和 Bing 站长工具（http://www.bing.com/toolbox/webmaster）。它们很容易设置，并且能给你非常重要的网站视图：主流搜索引擎如何看待它。只要网站的 robots.txt 文件有问题，有干扰良好搜索结果的 HTML 问题，有安全问题等任何问题时，它们都会发出警报。

22.1.7　监测分析

如果你的网站上没运行分析系统，那就从现在开始：它不仅能提供网站受欢迎程度的重要数据，还能告诉你用户是如何使用它的。Google Analytics（GA）很棒（并且还是免费的！），即便你的网站有额外的分析服务，也没有理由不把 GA 包含在内。你经常能在密切关注分析结果时发现细微的 UX 问题。某些页面没能达到你预期的访问量吗？那可能说明你的导航或推广有问题，或者是 SEO 的问题。你的跳出率高吗？那可能表明页面上的内容需要做些裁剪（人们是通过搜索到你网站上来的，但他们到了之后发现不是自己要找的东西）。除了 QA 检查列表，你还应该有个分析检查列表（甚至可以是 QA 检查列表中的一部分）。这个检查列表应该是个“活文档”；随着网站生命的延续，对你或者你的客户来说，什么内容最重要可能会发生变化。

22.1.8　性能优化

有多项研究表明，性能对网站流量有十分巨大的影响。这个世界节奏很快，人们希望他们的内容能快速传递，特别是在移动平台上。性能调优的首要法则是先分析，再调优。“分析”的意思是找出究竟是什么拖累了网站。如果你花了很长时间来加速内容的渲染，可实际上问题出自社交媒体插件，那你就是在浪费宝贵的时间和金钱。

谷歌 PageSpeed 是分析网站性能的好办法（并且现在 PageSpeed 的数据放到了 Google Analytics 中，所以你可以监测性能趋势）。它不仅能给出移动端和桌面端性能的总体评分，还能给出如何提升性能的优先级建议。

除非你现在有性能问题，否则可能没必要定期做性能检查（只要监测 Google Analytics，注意性能分值的显著变化应该就够了）。然而在性能提升后看到流量的大幅增长还是很令人欣慰的。

22.1.9　潜在用户追踪优先

在互联网上，访问者如果对你的产品或服务感兴趣，他们能给你的最强的信号就是留下自己的联系方式。你应该对这一信息极其关注。任何收集邮件或电话号码的表单都应该放在 QA 检查列表中做定期测试，并且在收集那些信息时应该总是有冗余。对于潜在客户来说，最糟的就是把收集到的信息又搞丢了。

因为潜在用户追踪对网站的成功非常重要，所以我向你推荐下面这 5 条信息收集的原则。

- *准备一个在 JavaScript 不行时的备用手段*
  通过 AJAX 收集客户信息很好，一般用户体验会更好。然而如果 JavaScript 不管出于什么原因不行了（用户可能把它禁用了，或者网站上的 JavaScript 可能有错误，导致 AJAX 不能正常运行），表单提交应该还可以用。要测试这个，可以禁用 JavaScript，用一下你的表单。用户体验不理想也没关系，关键是用户数据没丢。要做到这一点，即便你一般都是用 AJAX，也一定要在 `<form>` 标签中放一个有效并能用的 `action` 参数。

- *如果用AJAX，请从表单的`action`参数中获取URL*
  尽管不是绝对必要，但这可以防止你不小心忘记 `<form>` 标签中的 `action` 参数。如果你将 AJAX 绑到成功的无 JavaScript 提交上，就更不太可能丢掉客户的数据。比如说，你的表单标签可能是 `<form action="/submit/email" method="POST">`；然后在 AJAX 处理器中，你会这样：

  ```
  $('form').on('submit', function(evt){

  evt.preventDefault();
  var action = $(this).attr(\'action');
  /* 执行 AJAX 提交 */


  });.
  ```

- *最少提供一层冗余*
  你可能想要把线索保存到数据库中，或者是像 Campaign Monitor 这样的外部服务中。但如果数据库崩了，或者 Campaign Monitor 垮了，或者有网络问题了，怎么办？你

仍然不想丢掉那些线索。一种常见的冗余方法是除了把线索存起来，再发封邮件。如果采用这种方式，你不应该用个人邮箱，而是应该用共享的邮件地址（比如 dev@meadowlarktravel.com）：如果把邮件发给个人，则当那个人离开组织后，这个冗余就没了。你也可以把线索存到备份数据库里，甚至 CSV 文件中。然而，只要主存储失效了，就应该有某种机制向你发出警报。收集冗余备份只是这场战役的上半场，意识到会有失效的情况发生，并采取恰当的措施是下半场。

- *如果整个存储都失效了，通知用户*

  比如说你有三层冗余：主存储是 Campaign Monitor，如果它失效了，你备份到一个 CSV 文件中，并发送邮件给 dev@meadowlarktravel.com。如果所有这些通道都失效了，用户应该收的一条消息，说些“对不起，由于技术故障，请您稍后再试，或联系客服 support@meadowlarktravel.com”之类的话。

- *检查正向确认，而不是没有错误*

  让 AJAX 处理器在错误时返回一个带有 `err` 属性的对象，这是一种很常见的做法。这样客户端就会出现类似这样的代码：`if(data.err){ /* 将失效情况告知用户 */ } else { /* 感谢用户成功提交信息 */ }`。*不要使用这种方式*。设置一个 `err` 属性的做法没错，但如果 AJAX 处理器出错了，会导致服务器以响应码 500 做出响应，或者响应的不是有效的 JSON，*这样这种方式就会悄无声息地失效*。用户线索就凭空消失了，他们也不会知道。相反，为成功的提交提供一个 `success` 属性（即便主存储失效了：如果用户的信息通过什么方式记录下来了，你仍然可以返回 `success`）。这样客户端代码就变成了 `if(data.success){ /* 感谢用户成功提交信息 */ } else { /* 将失效情况告知用户 \*/ }`。

22.1.10　防止出现“不可见的”错误

我总能见到这种情况：因为着急，开发人员会用从来不会检查的方式记录错误。不管是日志文件、数据库中的表、客户端控制台日志，还是发送给僵尸地址的邮件，结果都是一样的：*网站有注意不到的质量问题*。要对抗这个问题，最好的防御措施是*提供一个易用的、标准的错误记录方法*。把它记录在文档中。不要搞得很难。不要搞得很模糊。确保每个接触到项目的开发人员都了解它。它能简单得像输出一个 `meadowlarkLog` 函数（`log` 一般被其他包用了）。这个函数把错误记录到数据库、普通文件、邮件，或者某种组合中都没关系：重要的是标准化。它还能让你提升你们的日志机制（比如在服务器扩展后，普通文件就不太实用了，所以你要修改 `meadowlarkLog` 函数，让它把日志记录到数据库中）。日志机制只要到位，就要把它记到文档中，确保团队中的所有人都了解它，将“检查日志”加到 QA 检查列表中，还要有如何检查的指导说明。

22.2 代码重用及重构

我一直能见到重新发明轮子的悲剧，一次又一次。一般只是些小事儿：有趣的是觉得重写比到几个月前做的项目里挖还要容易。所有这种小的重写工作都累积起来。更糟的是它还会传播到优秀的 QA 中：你可能不准备费事为所有那些小块代码写测试（如果你写了，就是比重用已有代码浪费了双倍的时间）。每个代码片段——做着相同的事情——有不同的 bug。这是个坏习惯。

用 Node 和 Express 开发有解决这个问题的好办法。Node 引入了命名空间（通过模块）和包（通过 npm），Express 引入了中间件的概念（通过 Connect）。有了这些工具，开发可重用的代码更容易了。

22.2.1 私有npm库

npm 公共库是保存共享代码的好地方；npm 毕竟就是为了这个而设计的。除了简单的存储，你还有版本，以及在其他项目中包含那些包的便利方法。

然而这里有个美中不足的地方：除非你在一个完全开源的组织中工作，否则你可能不想给所有可重用代码都创建成 npm 包。（除了知识产权的保护还有其他原因：你的包可能是专门针对组织或项目的，把它们放到公共库里没有意义。）

私有 npm *库*可以解决这个问题。搭建一个私有 npm 库可能要费些功夫，但确实有可能。

创建私有 npm 存储库最大的障碍是 npm 目前还不能从多个存储库中拉取 npm 包。所以说，如果你的 package.json 文件中混合着来自公共 npm 库的包（一定会有的）和私有库中的包，npm 会失灵（如果指定公共库，则无法获取到私有依赖项；如果指定私有库，则无法得到公共依赖项）。npm 团队说他们没有实现这一特性的资源（参见 https://github.com/npm/npm/issues/1401），但还有其他办法。

解决这个问题的办法之一是复制整个公共 npm 库。如果你觉得这既艰巨又昂贵（从存储、带宽和维护角度来说），没错，你是对的。更好的办法是提供一个到公共 npm 库的代理，让它将对公共包的请求转发到公众库，而私有包从它自己的数据库中提供。幸运的是正好有这样一个项目：Sinopia（https://github.com/rlidwka/sinopia）。

Sinopia 的安装极其容易，除了支持私有包，它还为你的组织提供了一个方便的缓存。如果你选了 Sinopia，应该知道它是用本地文件系统存储私有包的：你肯定想把包目录加到你的备份计划中去！ Sinopia 建议给本地包加上前缀“test-”。如果你为自己的组织创建私有库，我建议你用组织名称作为前缀（meadowlark-）。

因为 npm 的配置只能支持一个存储库，一旦“切换到”Sinopia（用 `npm set registry` 和

npm adduser），你就不能再用 npm 公共库了（除非通过 Sinopia）。要切回到 npm 公共库，或者用 npm set registry https://registry.npmjs.org/，或者直接删掉 ~/.npmjs。如果你想把包发布到公共库，则必须要这样做。

更简单的办法是用托管的私有库。Nodejitsu（http://www.nodejitsu.com）和 Gemfury（http://www.gemfury.com）都提供私有 npm 库。可惜这些服务都太贵。Ninjitsu 的服务价格从 25 美元 / 月起，并且只能提供 10 个包。要提高可管包的数量（50），需要 100 美元 / 月。Gemfury 的价格差不多。如果预算有限，这肯定不是个理想的选择。

22.2.2　中间件

就像我们在整本书里见到的，编写中间件不是什么巨大的、可怕的、复杂的事情。我们在本书中已经做过很多次了，过一段时间之后，你甚至都不用想就能写。然后下一步，是把可重用的中间件放到包中并放在 npm 库里。

如果你发现中间件的通用性比较弱，不足以放到可重用包中，则应该考虑重构中间件，让它可配置，变得更加通用。记住，你可以将配置对象传进中间件里，让它们适用整个情况。下面是在 Node 模块中输出中间件最常见的办法。接下来的所有办法都假定你将这些模块输出为一个包，并且那个包叫 meadowlark-stuff。

1. 模块直接输出中间件函数

如果中间件不需要配置对象，用这个方法：

```
module.exports = function(req, res, next){
    // 中间件在这里……记得调用 next() 或 next('route')
    // 除非这个中间件是终点
    next();
}
```

使用这个中间件：

```
var stuff = require('meadowlark-stuff');

app.use(stuff);
```

2. 模块输出返回中间件的函数

如果中间件需要配置对象或者其他信息，用这个方法：

```
module.exports = function(config){
        // 如果没有传入配置对象
        // 一般会创建一个：
        if(!config) config = {};

        return function(req, res, next){
            // 中间件在这里……记得调用 next() 或 next('route')
```

```
            // 除非这个中间件是终点
            next();
        }
}
```

使用这个中间件：

```
var stuff = require('meadowlark-stuff')({ option: 'my choice' });

app.use(stuff);
```

3. 模块输出包含中间件的对象

如果要输出多个相互关联的中间件，用这个办法：

```
module.exports = function(config){
        // 如果没有传入配置对象
        // 一般会创建一个：
        if(!config) config = {};

        return {
                m1: function(req, res, next){
                    // 中间件在这里……记得调用 next() 或 next('route')
                    // 除非这个中间件是终点
                    next();
                },
                m2: function(req, res, next){
                        next();
                }
        }
}
```

使用这个中间件：

```
var stuff = require('meadowlark-stuff')({ option: 'my choice' });

app.use(stuff.m1);
app.use(stuff.m2);
```

4. 模块输出对象构造器

这可能是最少见的中间件返回方法，但如果中间件非常适合用面向对象的方式实现，这个方法就比较好用。这也是实现中间件最需要技巧的方式，因为如果你将中间件输出为实例方法，它们就不会被 Express 的对象实例调用，所以 `this` 就不是你想要的实例。如果你想访问实例的属性，请参见 `m2`：

```
function Stuff(config){
        this.config = config || {};
}

Stuff.prototype.m1 = function(req, res, next){
        // 注意：'this' 不是你想要的实例；不要用它
```

```
        next();
};
Stuff.prototype.m2 = function(){
        // 我们用 Function.prototype.bind 将这个实例
        // 关联到 'this' 属性上
        return (function(req, res, next){
                // 现在 'this' 是 Stuff 实例了
                next();
        }).bind(this);
 );

module.exports = Stuff;
```

使用这个中间件：

```
var Stuff = require('meadowlark-stuff');

var stuff = new Stuff({ option: 'my choice' });

app.use(stuff.m1);
app.use(stuff.m2());
```

注意，我们可以直接在中间件 m1 中链接，但我们必须调用 m2（然后它会返回我们可以链入的中间件）。

22.3　小结

在你构建网站时，焦点时刻通常是网站推出的时候，也应该是这样的：围绕推出有很多兴奋点。然而，如果网站推出后没有悉心维护，客户在网站推出时产生的那股高兴劲儿很快就会被不满意的情绪取代。像推出网站那样悉心推进你的维护方案，能让客户得到良好的体验，这样他们会重复访问你的网站。

第23章

其他资源

我试图在本书中给出用 Express 构建网站的全面概述。我们也涉及了大量的基础知识，但对于你能得到的包、技术和框架而言，这仍然只是冰山一角。本章会向你介绍到哪里获取更多的资源。

23.1 在线文档

对于 JavaScript、CSS 和 HTML 文档而言，无人能与 Mozilla 开发者网络（MDN，https://developer.mozilla.org/）相媲美。如果需要查阅 JavaScript 文档，我或者直接在 MDN 上搜索，或者在搜索查询中加上“mdn”。否则 w3schools 肯定会出现在搜索结果中。负责 w3schools 搜索引擎优化工作的是个天才，但我建议你避开这个网站，因为我发现它的文档经常严重匮乏。

尽管 MDN 有优秀的 HTML 参考资料，但如果你刚接触 HTML5（或者即便不是），都应该看看 Mark Pilgrim 的《深入 HTML5》（http://diveintohtml5.info/）。WHATWG 维护着一个卓越的 HTML5 规范“活标准”（https://developers.whatwg.org/），如果我遇到实在难以回答的 HTML 问题，一般会首先找它求助。最后，HTML 和 CSS 的官方规范在 W3C 网站（http://www.w3.org/）上；上面的文档晦涩难懂，但有时候遇到非常困难的问题，它是你唯一的资源。

JavaScript 遵循 ECMA-262 ECMAScript 语言规范（http://www.ecma-international.org/publications/standards/Ecma-262.htm）。下一版 JavaScript 被称为 ES6（代号 Harmony），它的相关信息可以在 http://people.mozilla.org/~jorendorff/es6-draft.html 找到。要追踪 Node

（及各种浏览器）对 ES6 特性的支持，请参见 @kangax 维护的优秀指南（http://kangax.github.io/compat-table/es6/）。

jQuery（http://api.jquery.com/）和 Bootstrap（http://getbootstrap.com/）都有极其优秀的在线文档。

Node 文档（http://nodejs.org/api/）非常好，并且很完备，应该是 Node 模块（ 比如 `http`、`https` 和 `fs`）权威文档的首选。Express 文档（http://expressjs.com/）十分不错，但可能不太完备。npm 文档（https://npmjs.org/doc）既完备又实用，特别是关于 package.json 文件那一页（https://npmjs.org/doc/json.html）。

23.2 期刊

你绝对应该订阅下面这三份免费的期刊，并且每周都要认真阅读。

- JavaScript 周刊（http://javascriptweekly.com/）
- Node 周刊（http://nodeweekly.com/）
- HTML5 周刊（http://html5weekly.com/）

这三份期刊会为你推送最新的新闻、服务、博客和教程。

23.3 Stack Overflow

很可能你已经用上 Stack Overflow（SO）了，因为它 2008 年就出现了，并且已经成了最主要的在线 Q&A 网站，是获得 JavaScript、Node 和 Express（及本书涉及的所有技术）的问题答案的最佳在线资源。Stack Overflow 是由社区维基于声望的 Q&A 网站。声望模型决定了网站的质量和它持续的成功。用户的问题或答案被"投支持票"或答案被接受时，可以获得声望点数。提问不需要有声望，注册也是免费的。然而，你可以按照一种实用的方式来做事，以便提高你的问题得到解答的可能性，我们会在本节讨论。

声望在 Stack Overflow 上就像货币一样，尽管有些人是真的想帮你，但对于优秀的解答者而言，如果还有机会能获取声望，那就更是锦上添花的好事儿了。SO 上有很多非常聪明的人，他们都争着要第一个给出最佳答案（感谢 SO 对快速给出坏答案有很强的抑制因素）。你可以通过下面这些手段提高得到优秀答案的机会。

- *成为一个了解 SO 的用户*

  观看 SO 教程（http://stackoverflow.com/tour），然后阅读"如何问一个好问题？"（http://stackoverflow.com/help/how-to-ask）。如果你愿意，可以通读所有的帮助文档（http://stackoverflow.com/help），全部读完后你将会得到一枚奖章！

- 不要问已经回答过的问题

  尽职调查，试着找找是不是已经有人问过你的问题。如果你问了一个在 SO 上很容易找到答案的问题，你的问题很快会作为重复的问题关闭，人们经常会为此给你投反对票，这会对你的声望产生负面影响。

- 不要让人替你写代码

  如果你只是问“我怎么做某件事？”，你的问题很快就会被关掉，并被人投反对票。SO 社区希望你在向 SO 求助前自己先努力尝试着解决它。在你的问题里描述你尝试过的办法，以及为什么不行。

- 一次问一个问题

  一次问 5 件事情的问题：“我怎么做这件事，然后是那件，然后另一件事情，以及什么是做这个的最好办法？”这很难回答，并且会让人望而却步。

- 为你的问题作一个最精简的例子

  我回答过很多 SO 问题，当我看到有 3 页代码（或者更多！）的问题时几乎总是跳过去。把 5000 行代码的文件直接贴到 SO 的问题里不利于得到答案（但总有人这么干）。这是一种经常得不到回报的懒惰行为。这不仅让你不太可能得到有用的答案，并且也正是消除无关因素的过程会引导你自己解决问题（这样你甚至不用在 SO 上问这个问题了）。制作最精简的例子对你的调试技能有好处，对你认真思考问题的能力也有帮助，并且会让你成为 SO 上的好用户。

- 学会 Markdown

  Stack Overflow 用 Markdown 作为问题和答案的格式。格式良好的问题得到回答的机会也更高，所以你应该花时间学习这种实用并且越来越广泛的标记语言。

- 接受答案并投出赞成票

  如果有让你满意的答案，你应该接受它并投出赞成票。这样能提升解答者的声望，而声望是 SO 的驱动力。如果有多人给出了可接受的答案，你应该选出你认为最好的答案并接受它，然后给其他所有你觉得可以接受的答案投赞成票。

- 如果你在别人给出答案之前自己解决了问题，那就自己回答那个问题

  SO 是社区资源，你遇到的问题很可能其他人也会遇到。如果你已经解决了，本着助人为乐的精神，把你的答案放上去。

如果你乐于帮助社区，可以考虑自己回答问题。这既有趣又能得到回报，并且能带来比声望值更实际的回报。如果你的问题超过两天还没有人给出可用的答案，你可以在那个问题上用你自己的声望进行悬赏。声望会马上从你的账号上扣掉，并且是不可退的。如果有人给出了令你满意的答案，并且你接受了这个答案，那个人就会收到赏金，赏金最低为 50 个声望点。尽管提出有品质的问题也能获取声望点，但一般提供高品质的答案能更快地获

取声望点。

解答问题也是一种很好的学习方式。我一般觉得通过回答别人的问题学到的东西，比通过我的问题得到解答学到的东西多。如果你想真正彻底地学一个技术，学完基础知识后就开始解决 SO 上的问题吧。一开始你可能总是会被已经成为专家的人打败，但过不了多久，你就会发现自己也成了专家。

最后，你应该毫不犹豫地用自己的声望进一步发展你的职业生涯。一个良好的声望绝对可以放到简历上。最起码这对我来说有效，我现在的职位让我有机会亲自面试开发人员，我总是会被良好的 SO 声望打动（我觉得超过 3000 就是“良好的”SO 声望；5 位数的声望点很棒）。良好的 SO 声望让我知道这个人不仅能胜任自己的工作，并且还能清楚地沟通，并且一般都乐于助人。

23.4 为Express做贡献

Express 和 Connect 是开源项目，所以谁都可以提交“拉请求”（Github 术语，意思是你希望将你做的修改放到项目中）。但这并不容易：做这些项目的开发人员都是高手，并且在他们自己的项目上有绝对的权威。我不是给你泼冷水，只是说你必须付出很大的努力才能成为成功的贡献者，并且你不能随随便便地提交。

贡献流程很容易：你把项目分叉到自己的 Github 账号下，克隆那个分叉，做出修改，把它们推回到 GitHub 上，再创建一个拉请求，然后就会有项目里的人审查。如果你的提交很小，或者是一个 bug 修订，你可能很幸运地提交成功。如果你试图做些大的改变，你应该找个主要开发人员沟通一下，讨论你的贡献。你肯定不想在花费了几个小时或几天时间做完一个复杂功能之后，才发现它不符合维护者的愿景，或者已经有其他人在做了。

另一种为 Express 和 Connect 的开发做贡献的办法（间接地）是发布 npm 包，特别是中间件。发布你自己的中间件不需要别人批准，但你也不应该胡乱用低质量的中间件给 npm 库添乱：规划、测试、实现、写文档，你的中间件将会更成功。

如果你确实要发布自己的包，最少应该做下面这些事。

- *包名*
  尽管包的命名取决于你，但显然你不能挑一个已经被占了的名称，所以这有时候可能会比较难。不像 GitHub，npm 包不是按账号确定命名空间的，所以命名是全局性竞争。如果你正在写中间件，常规做法是在包名前加上前缀“connect-”或“express-”。不管这个包是做什么的，直接取一个朗朗上口的包名也没关系，但如果包名能提示它是做什么的会更好（有一个朗朗上口又恰当的包名示例叫作 zombie，它是用来模拟无头浏览器的）。

- 包介绍

  包介绍应该短小、精炼并且能说明包是做什么的。人们搜索包时，这是被索引的主要域，所以它最好能说明包是做什么的，而不是花言巧语（别担心，文档里还有地方让你展示自己的聪明才智和幽默风趣）。

- 作者 / 贡献者

  给他们应有的荣誉。继续。

- 许可

  这经常被忽略，并且没有什么比碰到一个没有许可的包更令人沮丧的了（你不确定能否把它用在自己的项目中）。不要做那种人。如果你不想对代码如何使用做什么限制，可以选 MIT。如果你想让它是开源的（并保持开源），另外一个流行的选择是 GPL。把许可文件放在项目根目录下是明智之举（应该用 LICENSE 开头）。要达到最广泛的覆盖范围，用 MIT 和 GPL 双许可。要看这个在 package.json 和 LICENSE 文件中的例子，请参见我的 `connect-bundle` 包。

- 版本

  为了让版本系统生效，你需要确定包的版本。注意，npm 的版本跟代码库中的提交号是分开的：你可以随意更新代码库，但人们用 npm 安装包时得到的东西不会变。你需要增长版本号，并重新发布，你的修改才能体现在 npm 库中。

- 依赖项

  你应该努力控制包的依赖项。我不是建议你总是重新发明轮子，但依赖项会让包变大，还会增加复杂性。最起码你应该确保不需要的依赖项不会出现在你的列表中。

- 关键字

  除了描述，关键字是让人们找到你的包的另一个重要元数据，所以请你选择恰当的关键字。

- 代码库

  你应该有一个。GitHub 是最常用的，但其他的也可以。

- README.md

  Markdown（http://daringfireball.net/projects/markdown/syntax）是 GitHub 和 npm 文档的标准格式。它是一种容易的、像 wiki 一样的语法，你很快就能学会。如果你想让人使用你的包，高质量的文档至关重要：如果我遇到一个没有文档的 npm 包，我一般不会再做进一步研究，直接跳过。最起码你应该介绍基本用法（有示例）。如果文档中介绍了所有选项会更好。如果还介绍了如何运行测试就是更进一步了。

当你准备好发布自己的包，这个过程是很容易的。免费注册一个 npm 账号（https://www.npmjs.org/signup），然后按以下步骤操作。

(1) 输入 `npm adduser`，然后用你的 npm 凭证登录。

(2) 输入 `npm publish` 发布你的包。

就是这些了！你可能想要从头开始创建一个项目，并用 `npm install` 测试你的包。

23.5 小结

我真诚地希望本书能给你所需的所有工具，开始使用这个振奋人心的技术栈。在我的职业生涯中，还从没有哪种新技术让我觉得如此心潮澎湃（尽管 JavaScript 做主角很奇怪），并且我希望自己呈现出了这个技术栈的优雅和希望。尽管我已经专职做网站很多年了，但我觉得，我要感谢 Node 和 Express，它们让我对互联网工作有了更深的、之前从未有过的认识。我相信它是真正能够提高认识的技术，而不是想把细节都隐藏起来，同时还提供了一个可以让你快速高效构建网站的框架。

不管你是 Web 开发新手，还是刚接触 Node 和 Express，我都欢迎你加入 JavaScript 开发者的行列，我期待着在用户组和会议上见到你，更重要的是，见到你做的东西。

关于封面

本书封面上的动物是一只黑百灵（百灵属）和一只白翅百灵（百灵属）。这两种鸟都部分迁徙，并且已知它们的活动范围远远超出了最适合的栖息地——哈萨克斯坦大草原和俄罗斯中部。除了繁殖，雄性黑百灵也会在哈萨克斯坦大草原过冬，而雌性则向南方迁徙。白翅百灵在冬天则向西部和北部飞得更远，越过黑海。这些鸟在全球分布得更为广泛，其中欧洲的白翅百灵占全世界总数的四分之一到二分之一，黑百灵则只占总数的百分之五到四分之一。

之所以叫黑百灵，是因为雄性的身体几乎全被黑色覆盖。雌性则只有腿和翼下的羽毛是黑色的，其他部分均呈深灰色或浅灰色。

白翅百灵翅膀上的羽毛呈独特的黑、白、栗三色，背部呈灰色，下体呈淡白色。雄性在外表上跟雌性的唯一区别是它有栗色的头冠。

黑百灵和白翅百灵的叫声都十分婉转悠扬，几百年来满足了喜欢各种百灵鸟的作家和音乐家的想象。这两种鸟成年时都以昆虫和种子为食，并且都是在地面上筑巢。人们曾观察到黑百灵将牛粪带到巢中用来垒墙或者铺路，不过其做出这种行为的原因尚不可知。

O’Reilly 用在封面上的很多动物都是濒危物种，它们全都对这个世界很重要。如果你想帮助它们，请访问 animals.oreilly.com/。

封面图片源自 Lydekker 所著的 *The Royal Natural History*。

关于作者

Ethan Brown 是 Pop Art 公司的高级软件工程师。Pop Art 是波特兰市的一家交互式营销机构，Ethan Brown 在那里负责网站及 Web 服务的架构和实现，他们的客户既有小企业，也有跨国公司。他有 20 多年的编程经验，包括嵌入式开发和 Web 开发，并且他相信 JavaScript 技术栈是未来的 Web 平台。

O'REILLY®

Node与Express开发

本书涵盖Express 4.0，系统地讲解了利用Express（Node/JavaScript开发栈中的重要组件）开发动态Web应用的流程和步骤。作者Ethan Brown通过开发一个示例应用讲授了开发公共网站和REST API的基础知识。此外还介绍了设计与实现Web架构的最佳实践，帮你用Express构建单页、多页以及混合Web应用。

Express在根本没有框架和有一个健壮的框架之间找到了平衡，让你自由选择架构。通过本书，熟悉JavaScript的前端和后端工程师会发现一种新的Web开发视角。

通过阅读本书，你将能够：

- **为渲染动态数据创建网页模板系统**
- **探究请求对象和响应对象、中间件及URL路由**
- **模拟用于测试和开发的生产环境**
- **实现文档数据库的持久化，特别是针对MongoDB**
- **让其他程序可以通过REST API访问你的资源**
- **用HTTPS、认证和授权开发安全的Web应用**
- **集成社交媒体、地理位置服务和其他第三方服务**
- **实现应用的启动和维护计划**
- **学习重要的调试技能**

Ethan Brown 美国俄勒冈州Pop Art公司的高级软件工程师，负责网站及Web服务架构的设计与实现。拥有20多年编程经验，从事过嵌入式开发和Web开发，他相信JavaScript技术栈是未来的Web平台。

“这本书信息量很大，同时又引人入胜。这是我见过的对这一主题的最佳介绍，它覆盖的内容范围非常之广，囊括了用Node与Express框架搭建具备生产能力的Web应用所需掌握的一切。”

——**Semmy Purewal**
Netflix公司高级软件工程师

“关于Node这一主题，我自己写过9本书。我可以证明这是一本非常不错的同题材书籍！”

——**Azat Mardanov**
资深软件工程师，
《JavaScript快速全栈开发》作者

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-38033-3
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks